AD-A259 144

AFIT/GCS/ENG/92D-11

ADA IMPLEMENTATION OF AN OBJECT DATA REPOSITORY

THESIS

Stephen Paul Perucca
Captain, USAF

AFIT/GCS/ENG/92D-11

DTIC
ELECTE
JAN 1 1993
S E D

196 p9
93-00190
012225

Approved for public release; distribution unlimited

93 1 04 099

AFIT/GCS/ENG/92D-11

ADA IMPLEMENTATION

OF AN

OBJECT DATA REPOSITORY

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science (Computer Science)

Stephen Paul Perucca, B.S. (Computer Science)

Captain, USAF

December 1992

Accesion For
NTIS CRA&I
DTIC TAB
Unannounced
Justification
By
Distribution /
Availability Codes
Dist | Avail and / or Special
A-1

Approved for public release; distribution unlimited

DTIC QUALITY INSPECTED 5

Preface

The purpose of this research was to develop a metamodel of the Rumbaugh et al. Object Modeling Technique (OMT). Rumbaugh's textbook, *Object-Oriented Modeling and Design*, was the primary source for this study. Because the OMT is used here at AFIT to teach students about object-oriented software engineering, it was selected above other object-oriented models as the model to be used for this thesis.

The essential data elements of the OMT were abstracted to form a data metamodel of the OMT, and the graphical elements were abstracted to form a drawing metamodel. With minor enhancements, the OMT captures all the essential elements of an object-oriented system. Because the data model developed for this thesis models all elements in the OMT, this data model provides an adequate tool to model any object-oriented system. Future research in this area, along with the evaluation of other object-oriented models, will provide valuable insight into the essential elements of an object-oriented system.

Many people helped me during this research effort. I would especially like to thank my thesis advisor, Dr. Thomas C. Hartrum, for his expert assistance and guidance while conducting my research. I am extremely indebted to him for his numerous hours of support. I would also like to thank my committee members, Major Paul D. Bailor (USAF) and Major Eric R. Christensen (US Army). Major Bailor provided me with a valuable preliminary structural design of the three sub-models of the OMT. Major Christensen provided me with expert Ada coding assistance. Additionally, Dan A. Zambon was always available to assistance me when I had problems with the hardware and software used during this thesis. Finally, I wish to thank my wife Nancy for her support, patience, understanding, and prayers during this tribulous time.

Stephen Paul Perucca

Table of Contents

List of Figures

Abstract

The many benefits of object-oriented software development such as encapsulation and extendibility have inspired numerous models of the object-oriented paradigm. Rumbaugh's Object Modeling Technique (OMT) is an object-oriented model that uses three submodels. The object, dynamic, and functional submodels of the OMT describe the data, behavioral, and processing aspects of a system by using entity-relationship, state-transition, and data flow models. Cross-links relate how the three submodels tie together. Two metamodels (models of models) of the OMT are developed using the OMT methodology and notation. The essential data elements of the OMT are abstracted into a data metamodel, and the graphical elements are abstracted into a drawing metamodel. Visual programming concepts and examples are briefly discussed. The OMT model is analyzed and designed using OMT object models. The data and drawing elements are modeled and implemented in standard Ada as object classes, associations, and aggregations. An object manager is developed to provide a generic core class, to maintain an object data repository, and to assert unique object identities. Instantiated examples (instance diagrams) verify the correctness of the metamodel designs. Problems encountered during development are discussed and recommendations are made to improve the OMT. Possible future research areas are presented.

ADA IMPLEMENTATION
OF AN
OBJECT DATA REPOSITORY

I. Introduction

Software experts have forecasted that object-oriented software development will impact the market in the 1990s much the same way that structured functional design did in the 1980s. Within five to ten years the object-oriented design (OOD) approach will dominate other approaches in the market (16:135). Object-oriented methodologies provide the software engineering techniques that these experts claim will improve the software development process. Object-oriented themes such as inheritance, classification or abstraction, and encapsulation enhance the development process by allowing extendibility, simplifying complexity, isolating affectability, and increasing reusability.

The intent of this thesis effort was to design and implement a modeling tool that adequately supports the analysis and design phases of software development. The tool allows designers to use the beneficial features of OOD to aid them in constructing models of their respective systems. The tool models the Object Modeling Technique (OMT) presented by James Rumbaugh in his book, Object Modeling and Design (17:4–6).

1.1 Background

Rumbaugh and other proponents of OOD suggest that almost any system can be developed using a good OOD model (17:1–11) (13:1+) (20:1–7). The methodology used by Rumbaugh, OMT, provides an excellent graphical notation to represent object-oriented concepts. The OMT uses three sub-models to describe object-oriented systems (17:6). These sub-models are:

- *Object Model* - "... describes the static structure of the objects in the system and their relationships. The object model contains object diagrams. An *object diagram* is a graph whose nodes are object *classes* and whose arcs are *relationships* among classes." This is basically an extended entity-relationship model (ERM).
- *Dynamic Model* - "... describes the aspects of a system that change over time. The dynamic model is used to specify and implement the *control* aspects of a system. The dynamic model contains state diagrams. A *state diagram* is a graph whose nodes are *states* and whose arcs are *transitions* between states caused by *events*." This is a basic state transition model (STM).
- *Functional Model* - "... describes the data value transformations within a system. The functional model contains data flow diagrams. A data flow diagram represents a computation. A *data flow diagram* is a graph whose nodes are *processes* and whose arcs are *data flows*." This is essentially a data flow model (DFM).

Rumbaugh states that "a complete description of a system requires all three models. The three models are orthogonal parts of the description of a complete system and are cross-linked (17:6)." He also claims that his OMT is applicable during all stages of system development, and increases in detail and complexity as development progresses (17:17).

Software design engineers have long used ERMs, STMs, and DFMs to assist in the high-level design of software systems. Such models can easily be decomposed into lower levels until the lowest levels map directly to their targeted implementation languages. Rumbaugh's OMT is well-suited for this type of decomposition.

Design engineers and system developers in many fields often use specific graphical notations to describe the systems they're building. For example, to describe their systems at the highest level, hardware designers draw "black-boxes" with input/output lines. These black boxes are further decomposed into smaller components such as computer chips. These chips are decomposed into logical gates, which are broken down into transistors, capacitors, resistors, etc. At the lowest levels,

these notational models can be "fed" into manufacturing machines to build the actual hardware. In the same manner, software engineers can use notational descriptions of their systems to represent its attributes and behavior. These descriptions can then be decomposed and "fed" into code generators to build the actual executable programs.

These visual representations of systems, or graphical models, can have a profound effect on engineers and designers. These models help designers to better understand their systems by giving them clear mental images of the design. This leads to better developed systems containing fewer design "bugs".

If a visual model is to have much value, it must be formalized (6:12). Encapsulation, connectedness, and adjacency should play central roles in the formalization. Lesser features, such as size, shape, and color, can also be exploited. Such features come complete with rigorous mathematical semantics, which will be of great value when analyzing the design correctness (testing for bugs) and automating the mapping to the target implementation language (code generation). After preliminary reviews of the OMT model by this author, it appears this model is adequately formalized to produce complete OODs.

Computer-aided drawing tools are often used by designers to help them build, manipulate, view, and store the systems they're building. These drawing tools are usually restricted to specific application domains, and they often contain only the most basic design notations and components. Such tools often lack the features that a true object-oriented design tool can provide such as:

- ***Extensibility*** - the ability to create new objects by extending the attributes and behavior of existing objects
- ***Abstraction*** - the ability to make an object easier to understand by focusing on the essential aspects of the object and ignoring the accidental properties
- ***Encapsulation*** - the ability to separate the external aspects of the object from the internal aspects by hiding information not needed by other objects

The value of these features to enhance the automated/assisted design tool was a major research issue in this thesis effort.

It is very difficult, if not impossible, to devise a model or tool that adequately addresses all aspects of a given set of problems. The goal of a model is to simplify the system description without making the model so enormous and complex that it becomes a burden instead of a help. The question then arises, "What do we do with the model when information can't be represented or is ambiguous?" Rumbaugh suggests the use of natural language or application-specific notation to adequately capture this information (17:18).

1.2 Problem Statement

The following questions were addressed and researched during this thesis effort:

1. Does the Rumbaugh OMT adequately represent an object-oriented system?
2. What are the essential data elements of the OMT?
3. Can this data information be adequately described, represented, and stored?
4. How and where are the three sub-models of the OMT cross-linked as stated by Rumbaugh?
5. What are the basic graphical or drawing elements of the OMT?
6. Can this drawing information be abstracted and separated from the data information?
7. How are these two descriptions of the OMT (data and drawing elements) combined to create a complete informational and graphical depiction of an object-oriented system?

1.3 Assumptions

The following assumptions were needed to help focus the research problem and effort:

- The OMT is a valid and effective methodology for an object-oriented representation of a software system. James Rumbaugh is an accomplished and well recognized expert in the field

of object technology. His OMT was developed at General Electric (GE) and is currently used at that corporation. GE's object technology consulting organization uses the OMT to help software developers in industry transition to object-oriented software development.

- Any system can be modeled by Rumbaugh's OMT described above. The OMT can be extended to capture items that can't be depicted in the OMT diagrams.
- The data and drawing aspects of the OMT can be extracted and separated. Afterwards, these elements can be recombined without loss of information.

1.4 Scope and Limitations

This research effort attempts to answer the problems addressed in the Problem Statement by designing and implementing data and drawing representations of the OMT. Designs used in previous research efforts (SATool, Abstract Model Manipulator, etc.) were reviewed, used, and modified as needed. This research effort does not actually build a graphical drawing tool, but it provides the underlying framework and elements needed to implement such a drawing tool in any graphical environment (i.e., MOTIF, MS-Windows, X-view).

1.5 Standards

During system design, the OMT methodology and notation was used to describe the data and drawing elements of the OMT. An abridged reference of the OMT can be found in Appendix A. Further discussion and examples are found in Rumbaugh's text (17).

During implementation, coding style followed the guidelines set in the Ada Style Guide (25). Other coding practices used were object-based design, encapsulated packaging, and generic code reuse.

1.6 Approach

The approach used to analyze and research this problem involved the following stages:

1. Researched object technology and object modeling topics. This was accomplished in a variety of ways.
    - Attended CSCE594 (advanced topics in software engineering) in the Department of Electrical and Computer Engineering at AFIT
    - Studied numerous object-oriented methodologies such as Mellor-Schlaer, Coad-Yordon, Booch, and Rumbaugh
    - Reviewed books, periodicals, conference proceedings, and reports on the subject
    - Became familiar with and used Object Maker to build object models (Object Maker is a commercial application for drawing object models for a variety of methodologies)
2. Defined the tool design requirements.
3. Designed and implemented a data model to describe the essential data information in the OMT.
4. Designed and implemented a drawing model to describe the OMT graphical notations.
5. Designed and implemented a "binding tool" to combine the data and drawing models.
6. Tested the modeling tool by creating and populating the models.
7. Reviewed the system design, implementation, and test results, and made or suggested any necessary changes or enhancements.

1.7 Equipment and Software

- **The hardware used for this thesis research was:**

- Sun SPARCstation 2 with color monitor

- The software used for this thesis research was:
  - SunOS (UNIX)
  - OpenWindows user interface environment
  - Verdix Ada compiler tools

1.8 Sequence of Presentation

This thesis is divided into six chapers. Chapter I introduces this research effort. Chapter II reviews current literature that is pertinent to this research. Chapter III presents and analyzes the requirements for this modeling tool. Chapter IV presents and explains the design of this tool, and Chapter V shows how the design was implemented. Finally, Chapter VI summarizes the results of this research and makes several recommendations for further work in this area.

II. Literature Review

Prior to building the data and drawing models of the OMT, a literature review was conducted to research topics that related directly or closely to this thesis problem. This chapter evaluates the OMT, discusses visual modeling, reviews existing visual modeling tools, and finally, presents methods to model other models.

2.1 Evaluation of the OMT

The OMT is not the only object-oriented analysis method available today. Since the mid-1980s, numerous object-oriented analysis models have emerged. Some of these are (8:21):

- 1985 - Edwards
- 1987 - Jacobson
- 1988 - Shlaer et al.
- 1989 - Bailin, Colbert, and de Champeaux et al.
- 1990 - Coad et al., Gibson, Rumbaugh et al., and Wirfs-Brock et al.
- 1991 - Kurtz et al. and Page-Jones et al.
- 1992 - Odell et al.

The Booch method is another very popular model. This section evaluates Rumbaugh's OMT by studying its features to determine some strengths and weaknesses. Chapter VI contains a section that discusses possible improvements to the OMT.

A comparitive analysis of the object-oriented methods listed above can be found in an article by de Champeaux (8). The following is a list of features that the article felt were important parts of any object-oriented method (8:31):

1. **Inheritance**
2. **Multiple inheritance**
3. **Attributes for objects**

4. Aggregation / part-of
5. Relations or associations
6. State transition diagrams
7. Events
8. Event traces
9. Triggers
10. Data flow diagrams
11. Global parallelism
12. Parallelism inside object
13. Tool
14. Integrated tool set

The OMT scored positive in all areas except triggers, tool, and integrated tool set. However, after further review, this author has found otherwise. First, triggers are described in the article as transitions that trigger other transitions via events (8:26). The OMT provides the ability and notation to send an event to another class. Since each class has its own dynamic model, recipient transitions of triggers can be easily located. Next, the developer of the OMT model, General Electric (GE), has developed a tool called OMTool. OMTool is "a computer-based graphical tool for the analysis and design of systems using object-oriented techniques" that allows creation and maintenance of object models and diagrams (11). Finally, an integrated tool set (extensions to OMTool) is currently under development at GE.

Rumbaugh's model allows an easy transition from the analysis phase, through the system design phase, and finally to the implementation phase of software development (17:5–6). The analyst can use the OMT to model the requirements and conceptual representation of the problem being studied. The system designer can use the same model and notation to add design considerations. Finally, the implementer can add additional implementation details. All along the way, new classes can be added as the problem is abstracted at the various levels. This feature or capability of the OMT model facilitates smooth transitions through the development process.

Object behavior characterization is often the most difficult aspect to model at the modeling level without going into implementation details (8:25). For this reason, it is often left until the implementation phase. To be adequate, models of behavior should capture both *what* the object does and *how* it is accomplished. The OMT allows the analyst to describe *what* the behavior is by giving it a descriptive name and narrative text. The system designer can extend behavior by using the object model to describe the interface, and by using the functional and dynamic models to describe the basic behavior. The implementer can then fill in *how* the behavior executes on the target system by adding code syntax.

The OMT provides a variety of ways to model the details of behavior. The dynamic model captures the *changes* and *transformations* occuring within objects by using a state transition notation. In the dynamic model, activities, actions, and events are used to depict object behavior. The functional model captures the object and data *processes* that occur by using a data flow notation. Processes are used in the functional model to describe the "sub-operations" of object behavior.

Throughout Rumbaugh's text, the author tries to describe how the three models relate and cross-link (17:110–111, 137–139, 229–230). Many connections between the models are straightforward, direct, and unambiguous. However, ambiguities exist in a few cases, and the author lets the reader decide how the models relate and connect. The following is a list of some of these cases:

- *Discovering operations* (17:183–185) - Operations can be found at various locations in the OMT sub-models. These locations are:

    1. Operations from the object model
    2. Operations from events
    3. Operations from state actions and activities
    4. Operations from functions (processes)
    5. Shopping list operations

If not modeled properly, behavior in one location can be ambiguous and conflict with behavior in other locations. Furthermore, "shopping list" operations are implied or suggested operations, and expert analysis is required to discover such operations.

- *Assigning responsibility and location for operations* - Sometimes it may not be clear where a particular operation needs to be placed. For example, the shopping list operations mentioned above eventually need to be encapsulated into existing classes, or new classes need to be created and given responsibility of these operations. Furthermore, "each function in the data flow diagram [functional model] corresponds to an operation on an object (or possibly several objects) (17:184)." Therefore, processes in the functional model may cross object boundaries.

Some object technology experts have charged that Rumbaugh's OMT is among a group of ***hybrid*** object methods that can exhibit ***object-slicing*** characteristics (2:26). When modeling an object-oriented system, the OMT method advocates building an information model first (object model), then adding behavioral (dynamic model), and finally processing (functional model) views. The result, as graphically shown in Figure 1, is a three-layered model describing the system under study. During this modeling process, the boundaries of an object's data, process, and behavior elements may become fuzzy.

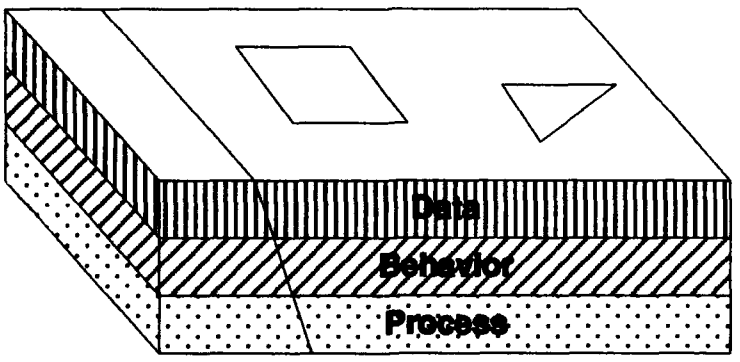


Figure 1. Layered Model

As defined by the author, object-slicing describes "the activity of creating objects by taking cross-sections of the multi-layered model (2:26)." Any way the layered models are sliced will result

in objects containing data, behavior, and process. Figure 2 graphically shows this slicing process. In order to assure that good objects result from this slicing, rigid criteria is needed to define where to slice the models. Therefore, the quality of objects created from the OMT method is determined by the object-slicing process. A major concern in this thesis effort was to adequately define this slicing process, namely, relating and cross-linking the object, dynamic, and functional models.

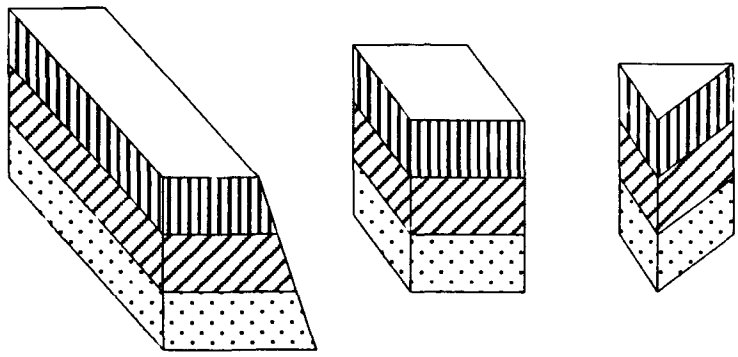


Figure 2. Object Slicing

2.2 Visual Software Development

As mentioned in the introduction to this thesis, software development is becoming more object-oriented (5:39). The highly graphical user interfaces becoming popular today such as MOTIF, OpenLook, and Windows[1] are well suited to the object-oriented paradigm. This visual appeal has inspired developers to create visually interactive methods of software development to replace the traditional iterative, textual process. Visual application builders are quickly emerging in the software development marketplace. "Many of these tools require no more than three days of training to use (14:15)." This section explains the important concepts of visual programming, and the next section presents some examples of currently available visual programming tools in the marketplace.

Figure 3 shows the population of software users and developers (5:40). There is a dividing line between those who use software and those who are able to build software. Traditionally,

[1] Windows^{TM} is a registered trademark of MicroSoft, Inc.

those above the line are software designers and coders, those who know programming languages and design principles. However, visual programming lowers this line and allows a wider domain of people to build their own software applications. The following subsections illustrate the basic components of visual programming, how visual programming works, and some of its benefits.

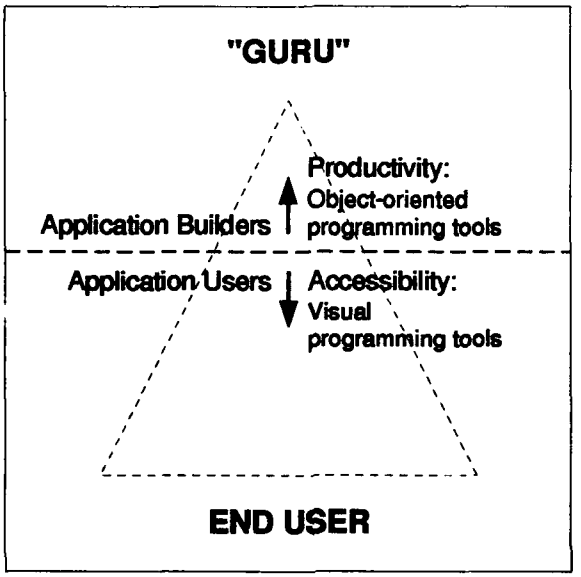


Figure 3. Software Programming Pyramid

2.2.1 Application Building Components In an object-oriented environment, the building blocks are objects or components. In a "pure" object-oriented language such as Smalltalk, everything is an object, even basic data types such as characters and integers. Other "hybrid" languages such as Ada and C++ define objects at a higher level such as encapsulated packages and class structures. In either case, these basic components are essential to build more useful, application-specific components.

The visual representation of these components must be intuitive to the person using them. At the lowest levels, components may be abstract items such as integers, loops, and lists. It may be necessary to be "creative" when designing the visual icons for such items. More complex objects such as buttons, windows, and clocks, are usually much easier to visually depict. Furthermore, the

inner characteristics or attributes of these components should be visually described when needed. For example, a digital clock should be visually discernable from an analog clock.

When building the interface to these components, very thoughtful consideration is required. The interface should be published, royalty-free, and standardized to encourage wide-spread reuse (5:40). Documentation of the components' behavior, performance, storage requirements, etc., helps designers "pick and choose" the right pieces to build their systems. Royalty-free interfaces allow developers to build components "around" non-existing components that will be purchased and added later. Finally, a standardized interface prevents unnecessary incompatibilites.

With a basic set of components whose characteristics are visually intuitive and whose interfaces are well-designed, it is possible to build larger, more complex components and sub-systems. Some of the more "reusable" high-level components can be added to the base component set. In this way, large, complete applications can be built from an expanding component inventory. The visual representation of this building process must also be intuitive. For example, plugs and sockets may be easy to understand and use for an electrical engineer who is building components on a "bread board." Network administrators who design and build network systems will benefit from connecting components or nodes by lines and arcs.

2.2.2 Application Building Environment A visual development environment requires a variety of integrated tools and pieces (5:41). Some of these tools and pieces are:

- *Component repository* - the component characteristics database. Besides storing the actual components, the database can store performance, reliability, interface, author, and other pertinent information about the objects. It also provides the necessary database management functions such as assess authorization, concurrent use, inventory control, and history tracking.
- *Exerciser* - the component test-bed. The exerciser can test component functionality, performance, timing, and other capabilities and features important to those evaluating or using the

components. This information can then be attached to the components and placed in the repository.

- *Encapsulator* - the component "packager." This tool allows application builders to use low-level language constructs such as loops, conditions, and procedures to describe the inner workings of the components. "Gurus" can use the encapsulator to fine-tune their components. With the use of the encapsulator, the application builder can then add a standard interface for external access to the component's attributes and behavior.
- *Integrator* - the component interconnector. The integrator provides the means to graphically build complex components or complete applications by selecting components from the repository and connecting them. This tool can be used by "non-programmers" and "gurus" alike by tailoring the component set and graphical interface to the skill and knowledge of the user.
- *Application builder* - the front-end tool. This tool combines and integrates all the other tools to form a complete, coordinated visual programming environment.

In an article about visual programming, Mandelkern presents his view on how a visual programming environment should look (5). Figure 4 illustrates this view. The tools described above are all shown in this figure.

2.2.3 Benefits of Visual Development Visual programming provides numerous benefits. The following list summarizes some of these benefits:

- Allows developers to build applications using the components and syntax with which they are most familiar with. The need to learn new syntax and terminology is reduced or eliminated.
- Encourages basic-component builders to design useful, standardized, high-performing, well-encapsulated, bug-free components. Such components are more likely to get used and/or purchased.

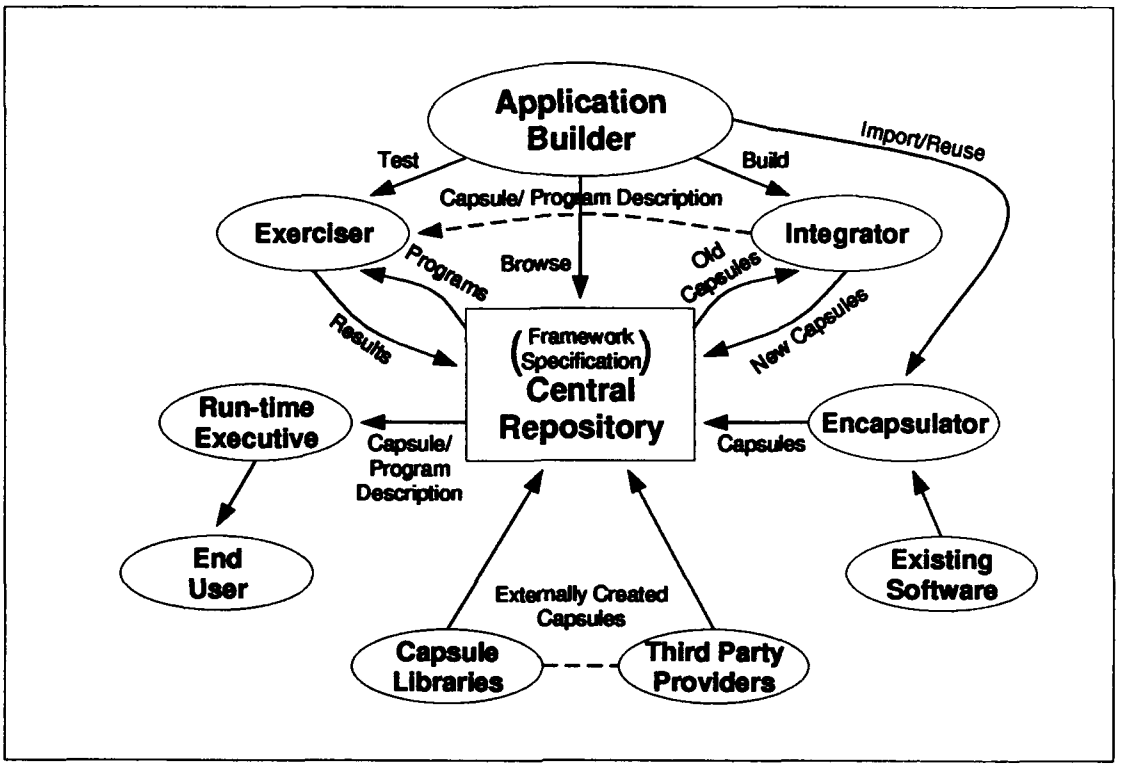


Figure 4. Visual Programming Environment

- Provides the means to allow more people ("non-programmers") to develop their own applications without relying on "gurus" to do it for them. Individuals no longer need to know a coding language such as Ada, Pascal, or C++ to write applications.
- Eases the transition from analysis, through design, to implementation by allowing programs to be represented at levels that are more conceptual than code. For example, the same models used during analysis and design can be used directly in the visual programming environment to build executable applications.
- Translates well to the object-oriented paradigm. Components are real-world entities, entities that can be easily implemented as objects.

2.3 Existing Visual Modeling Tools

This section looks at some visual design tools currently available in the personal computer (PC) market. Each has important object-oriented features that allow easy development of systems by using a graphical user interface (GUI). The intent of this review was to examine these tools and discover how they used object-oriented features. Five design tools were reviewed.

2.3.1 Look and Feel Kit Digitalk, the developer of Smalltalk for the PC, has developed a design tool for Smalltalk (an object-oriented programming language - OOPL) called the Look and Feel Kit (10). "It makes visual the concepts and mechanisms of OOP." Instead of coding programs in the traditional way by using textual statements, the kit allows programs to be created by selecting and positioning visual icons or "components" into a graphical editor window. These components can be connected to other components by lines or "wires". These wires add functionality to components by showing communication (message or parameter passing) between objects.

The Look and Feel Kit uses the benefits of inheritance in the following ways. First, the components and wires are categorized in a hierarchical structure. Components can have sub-components that depict more specialized or generalized components. For example, conditional and iteration loops can be grouped under the general category of loops. Conditional loops can be further specialized into pre-conditional (WHILE) and post-conditional (REPEAT - UNTIL). Wires are also of different types. Read wires show incomplete links (further information is needed before compiling), and green wires show completed links. The Look and Feel Kit also uses inheritance to allow component sets to be extended. Once completed, a complex component or program can be added to the component set and used in later programs. This extendibility of the component set allows complex programs to be built from a much simpler set of primitive components and programs.

"The determining factor in the Look and Feel development process is the quality and completeness of the component set." The set can be extended with applications created locally or with

components purchased from other designers. "With a thorough set of components, you should be able to construct non-trivial applications, relying mainly on visual programming."

2.3.2 Adept Developed by Symbologic Corporation, Adept uses visual, OOP techniques under a windowing environment to allow application developers to generate programs by drawing flowcharts and decision trees (15). The nodes of the decision trees come in different selectable styles and are connected by arcs. Arcs are color coded to distinguish their types.

Node attributes and behavior are partially extendable. Within the scope of the four kinds of nodes (start, work, case, end), nodes can be given text attributes (labels) to be displayed next to the node. Node behavior can be programmed using Adept script commands.

"Symbologic's Adept is ideal for building procedure-oriented applications that can be as simple as granting a personal loan or as complex as dismantling a nuclear warhead. Yet it's so completely easy to use that almost anyone can become a skillful creator of an expert system in very short order (15)."

2.3.3 Icon Author Icon Author, from Aim Tech Corporation, is a multi-media authoring tool that uses a flowchart metaphor to "make things easier (22)." Visual icons are used to depict the flowchart nodes, and directed lines connect the nodes to show flow and conditional branching. By selecting icons from the toolbox and dragging them to the work area, complex applications can be created without writing a line of code. A person can quickly map out the entire application flowchart and then go back later and fill in the details.

Nodes in Icon Author have attributes or "content." Once a node is instantiated by placing it in the work area, a dialog box appears and prompts you to provide the content. While debugging, nodes can also be temporarily disabled (if details haven't been filled in) in order to check various flowchart decision paths. This feature provides the benefits of encapsulation by hiding the content or "internal workings" from neighboring nodes.

Even though Icon Author is a specialized product for making professional multimedia applications, it's very capable of authoring other tasks. "Its intuitive, slick graphical interface can save hours of development time."

2.3.4 Object Vision Borland's Object Vision is a forms design tool that lets a person implement the logic behind forms (filling in the form blanks) by diagramming decision trees (23). Because decision trees can graphically depict IF-THEN-ELSE and CASE conditions, complex logic can be abstracted into a more understandable visual picture. Logic flow in Object Vision can easily be traced by following lines from the source, through subordinate decision nodes, to the destination.

2.3.5 Object Linking and Embedding (OLE) MicroSoft Corporation has developed and is currently using an OOD technique called object linking and embedding (OLE) (4). OLE works as follows: Document objects created in one application can be dynamically linked to and embedded in another, possibly totally different application. For example:

> "If your spreadsheet and your word processor both support OLE, you can annotate your worksheet with a text box containing words created in your word processor. To edit the text, you'll double-click the box; the word processor will appear. Quit the word processor, and your worksheet will automatically be updated. The text box, in this case is the embedded object. Going the other way, you can embed worksheet objects, such as tables or charts, in your word processor's documents."

OLE provides some of the benefits of extendibility by allowing easier creation of compound documents that share common objects.

2.4 Modeling the OMT Model

The main purpose of this thesis was to develop and implement both data and drawing model representations of the OMT. On page 229 of Rumbaugh's text, the author introduces the OMTool. He discusses how the OMTool was used to store both the logical and graphic representations of the OMT model. "The graphical model stores the picture that is drawn on the screen: the choice of

symbols, position of symbols, length of lines, and so forth. The logical model stores the underlying meaning of the picture, that is, classes, attributes, operations, and their relationships (17:229)." Chapter 10 of Rumbaugh's text uses examples taken from these two models of the OMTool. The examples provided in the text are few. Therefore, an effort was made to contact (telephone and electronic mail) the makers of OMTool at GE to obtain further examples of the graphical and logical models. No one at GE could or would provide the requested information (11).

While browsing the library periodicals, a very important article was found that related directly to this thesis effort (19). The article contained an abridged system design of the OMT model called the OMT metamodel. The article was published after the system design phase of this thesis was completed and while the implementation phase was close to being completed. It can be noted that this metamodel is very similiar to the data model that was designed for this thesis. Therefore, it is expedient to show this metamodel so that it can be compared to the data model developed for this thesis.

2.4.1 Metamodel for the Object Model Figure 5 is a metamodel that partially describes the data aspects of the object model portion of the OMT (19:14). An attribute has a name and can be either a link attribute, qualifier, object attribute, or discriminator. An association has a name and can have zero or more link attributes. A class has a name and can have zero or more object attributes. A role has a name, multiplicity (one, many, optional, or specified), and an optional qualifier. Associations have a role on each of their ends where they connect to a class. A class can connect to zero or more roles, roles which are attached to the end of an association. Finally, generalization can be concrete or overlapping and can have an optional discriminator. A generalization has one superclass and one or more subclasses.

2.4.2 Metamodel for the Dynamic Model Figure 6 is a metamodel that partially describes the data aspects of the dynamic model portion of the OMT (19:15). An event has a name and can have zero or more event attributes. A state has a name and can have an optional activity,

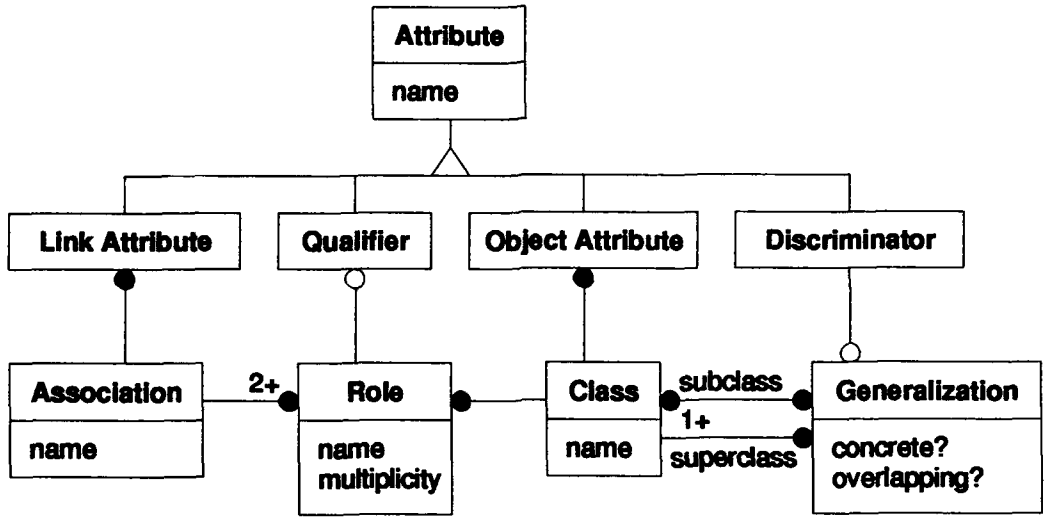


Figure 5. Partial Metamodel for the OMT Object Model

entry action, and exit action. An activity has a name and can be assigned to zero or more states. An action has a name and can be assigned to zero or more states and/or transitions. A condition has a condition string (boolean expression) and can be assigned to zero or more transitions. A transition connects a source state to a target state, and it can have optional events, guard conditions, and/or actions. An event generalization has one super-event and one or more subevents. A state generalization has one superstate and one or more substates.

2.5 Literature Review Summary

Rumbaugh's object modeling technique is an adequate object-oriented model that has been widely accepted by the object community. The OMT addresses all areas of an object-oriented environment such as inheritance, associations, aggregations, and attributes/behavior encapsulation. Because the OMT is composed of three submodels, it exhibits object-slicing characteristics. To create well-designed classes and objects, it is essential to correctly cross-link and relate the three submodels, especially in the area of behavior.

Visual programming is a "new wave" in software development. The object-oriented paradigm provides the methods needed to create visual application builders. A basic visual development

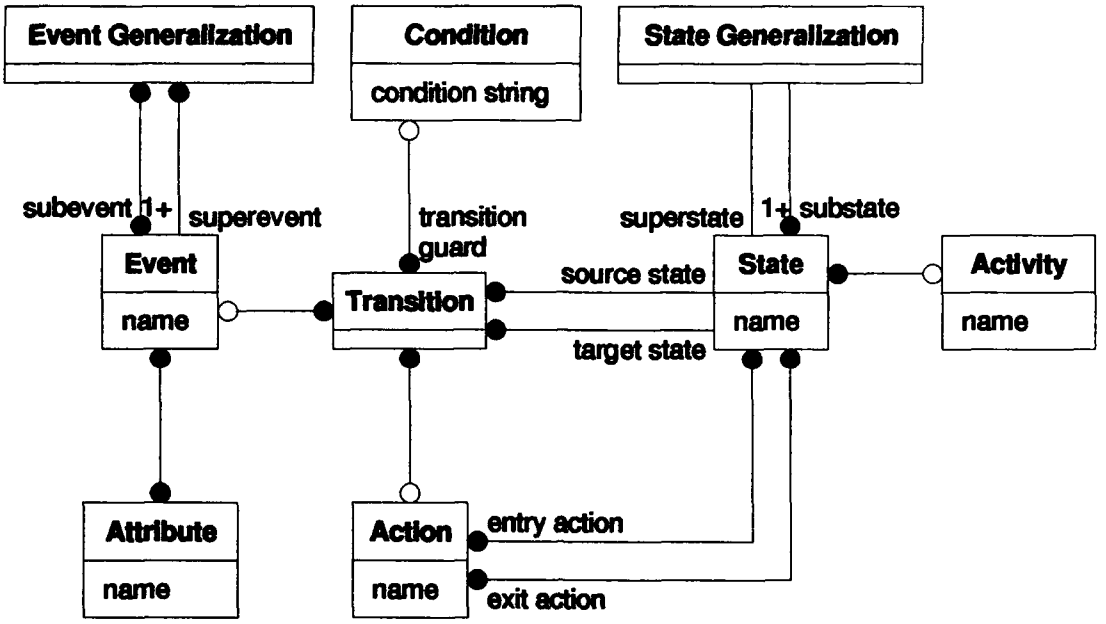


Figure 6. Partial Metamodel for the OMT Dynamic Model

environment is composed of a component repository and an integrator. Advanced tools include exercisers and encapsulators. A variety of visual modeling tools are currently available, and each exhibits many of the benefits of visual and object-oriented programming.

The main task of this thesis was to design and implement a metamodel of the OMT. The initial literature search provided very little direct support in this area. However, later searches revealed that GE, the creator of the OMT, has published abridged metamodels of their analysis of the OMT. GE's metamodel and this author's version are very similar. The next chapter presents the thesis requirements and the OMT design requirements.

III. Requirements Analysis

This chapter presents and analyzes the requirements for the design and implementation of the data and drawing modeling tool for the Rumbaugh Object Modeling Technique (OMT). These requirements were derived from the sponsor, the thesis advisor, and from Rumbaugh's text (17).

3.1 General Requirements

1. This tool must capture the essential data information (also called the metadata or logical data) of the OMT methodology. This portion will be called the data model.
2. This tool must capture the drawing information (also called the graphical data) of the OMT model notation. This portion will be called the drawing model.
3. The data and drawing models must be designed and implemented as separate systems. The information in one model should be mutually exclusive from the information in the other model.
4. The tool must allow initialization and population of the data and drawing models. Object persistance is required on secondary storage media.
5. All system design models of the tool must be done using the OMT methodology and notation. In other words, the tool design diagrams will be models of themselves.
6. All parts of the modeling tool must be implemented in Verdix Ada. Verdix Ada library extensions such as the VSTRINGS package will be allowed.
7. Proprietary software and software components should not be used if possible. Needed components such as generic LIST packages should be in the public domain.

3.2 General Analysis

Analysis of the general requirements suggest the following problems need to be solved:

1. The design and implementation of a data model to initialize and populate the object database or repository containing the essential logical information of the OMT methodology.
2. The design and implementation of a drawing model to initialize and populate the object database or repository containing the graphical information of the OMT notation.
3. The design and implementation of a "binding tool" to bind the above subsystems together and insure that the information in both models is mutually exclusive.

These three requirements define the major thrust of this research and development effort. The following three sections analyze these three areas in greater detail.

3.3 Data Model Requirements

As previously stated, the data model must capture the essential information of the object modeling technique found in Rumbaugh's text. A review of this text shows that the OMT is divided into three major sub-models and zero or more optional sub-models (17:6). (For a complete and condensed description of the OMT notation, see Appendix A of this thesis). The first three sub-models of the OMT are:

- ***Object Model*** - a modified entity-relationship model
- ***Dynamic Model*** - a basic state-transition model
- ***Functional Model*** - a data-flow model

All three models are necessary for a complete object-oriented description of a system. The models are cross-linked. The object model is the core of the OMT and describes *what* the system is composed of and *what* is changing and transforming. The other two models provide information that supports the object model and describes *when* and *how* the components of the object model are changing and transforming.

The optional models are any domain-specific models, notations, and languages needed to capture information about a specific domain that can't be captured in the above three models. The requirement for these optional models is suggested by Rumbaugh (17:18). However, this requirement was not met by this research effort because it is impossible to anticipate the specific modeling needs for every domain. However, because the data modeling tool was designed using object-oriented techniques, it should be possible to extend the tool and add domain-specific models.

The next three subsections discuss the data model requirements for each of the three major OMT sub-models listed above.

3.3.1 Object Model Data Requirements The object model of the OMT "provides the essential framework into which the dynamic and functional models can be placed (17:17)." The goal of the object model is to capture the essential elements and information of the application that are important for the domain under consideration. The essential data information portrayed in the object model can be abstracted and generalized into the following categories:

- Classes, and the objects instantiated from these classes (17:21–26)
- Associations and links between classes and objects (17:27–38)
- Generalization and inheritance of classes (17:38–43)
- Grouping constructs (17:43)

The requirements from each of these categories were reviewed and analyzed. These next four subsections describe these requirements in detail.

3.3.1.1 Class and Object Data Requirements The fundamental data elements of the object model are classes and the objects instantiated from these classes. Therefore, the data model portion of the modeling tool must adequately describe the data characteristics of the classes and objects in the OMT. The class entity is analyzed first.

A class encapsulates attributes and behavior into a single, identifiable entity. Each class has a name that is unique from other classes. For example, a class named BIKE might describe the characteristics and behavior of a bike while a CAR class might describe a car. The class name should be a noun.

A class may be either a concrete class or an abstract class (17:61–62). An abstract class is a superclass that cannot be directly instantiated while a concrete class can be directly instantiated. Abstract classes are often used to conveniently organize class hierarchy structures. Classes can also be divided into derived classes and base classes (17:75–76). Derived classes contain redundant information that can be derived from other derived or base classes. Base classes do not contain derived information. Derivation descriptions are used to show how and where derived classes originate. Derived classes are typically used to combine information that is a cross-section of two or more other classes, much like a "join" operation in a relational database is used to create a new table from existing tables.

Some classes might not have state or behavior. This is especially true early in system analysis when all the facts about classes aren't known. However, if a particular class is missing both attributes and operations when implementation begins, then the class is probably not needed. If it is needed, it should probably be changed to an association.

The attributes within a class describe the possible ranges and types of data information the class can possess. Any of these attributes may have initial values when an object is instantiated. Attributes also have names that are unique within the encapsulated class. Attribute types, ranges, and initial values might not be known during the early stages of analysis and design, and thus omitted. They can be added later during implementation. Some examples of class attributes are:

- **HardwareId**
- **NumberOfPages** : integer
- **SocialSecurityNumber** : string = "000-00-000"

Attributes are either derived or base attributes. Derived attributes originate their values from other derived and/or base attributes. Base attributes contain only actual values. A derivation string describes how and where derived attributes get their values. For example, the attribute AGE can be derived from BIRTHDAY as follows: AGE = TODAY - BIRTHDAY.

Attributes may have constraints. This becomes necessary when an attribute's type, range, and initial value don't adequately define or restrict the attribute. For example, a triangle is a polygon that is constrained to have only three vertices, and the sum of the lengths of any two sides of the triangle are greater than the length of the third side.

Class behavior is represented by operations (also called methods) within the OMT object model. Each operation has a name and a signature or interface. Together, the name and signature uniquely identify the operation from other operations in the class. A signature includes an optional argument list and an optional return value (also called a function return value). Each item in the argument list is a subroutine parameter that has a name and a type. The return value is a data type. The actual behavior portion (the *how* and *when*) of the operation can appear in the class in the form of a code block, or the behavior can be described in the dynamic or functional model. Behavior in the code block can be described by any notation or language that is convenient and familiar to the modeler.

Abstract operations are used as "place-holders" in superclasses to show common operations that exist within subclasses. These subclasses provide the concrete operations that implement the actual behavior. For example, the superclass POLYGON has subclasses TRIANGLE and RECTANGLE. The operation POLYGON.DRAW defers execution to either TRIANGLE.DRAW or RECTANGLE.DRAW, depending on the actual instantiated subclass.

Some attributes and operations may be class attributes and class operations (17:71). Class attributes describe values common to an entire class such as defaults, maximum values, and sets.

Class operations are operations on the class itself such as creating object instantiations and getting information about the set of instantiated objects.

Objects are instantiated from classes. When instantiated, the object possesses the name and all the characteristics and behavior of the class. However, each object must be uniquely identifiable from all other instantiated objects. Even if two or more objects have the same attribute values, they must be distinctly and uniquely identifiable. For example, if two objects of the same class are instantiated at the same time, initially their attributes are the same. Therefore visible attributes alone cannot be used to distinguish separate objects. Identity is essential to objects.

Although an instantiated object can set its attributes to any allowed value within the predefined type and range, it must not modify the operations it inherited from the class. If modifications to operations are needed, new classes must be formed, and the objects must be instantiated from these new classes. Therefore, operations may be shared between objects within the same class.

3.3.1.2 Association and Link Data Requirements "Links and associations are the means for establishing relationships among objects and classes (17:27)." These relationships provide the "glue" to connect encapsulated classes and objects to create new and more complex classes, objects, and systems. They provide the vehicle for objects and classes to communicate. Therefore, it is essential to allow these relationships in the object model. The OMT object model supports such features.

A link is merely an instantiated association, a connection between object instances. Like objects, links inherit the features of the association from which they are instantiated. Furthermore, each instantiated link must be uniquely distinguishable from all other links.

Associations describe a group of links that have common structure and common semantics (17:27). Their name is a verb that describes the purpose of or work performed by the relationship between the connected objects. This verb often implies a directed relationship. This is the *forward* direction. The opposite direction is the *inverse* direction. For example, in a WORKS-FOR

association between the classes COMPANY and PERSON, the person is the forward class, and the company is the inverse class.

Associations may be binary, ternary, or of higher order. This tool will allow either binary or ternary associations. Since higher order associations are very uncommon and hard to draw and implement (17:28), they aren't required in this modeling tool.

Links connect one object to one another object. Multiplicity is one-to-one. In associations, multiplicity at any connection to a class can be either one, optional (zero or one), many (zero or more), or numerically specified (i.e., 1+ or 0,2-4). When the many multiplicity is used, it is often necessary to further describe or reduce this multiplicity. Ordering, such as sorted alphabetically, can be used. A qualifier can also be used on an association to describe and reduce the multiplicity at the connection. For example, a DIRECTORY may have many FILEs, but a FILE belongs to only one DIRECTORY (17:36). A FILE NAME qualifier can be placed at the DIRECTORY side of this association to clarify and reduce the multiplicity to one-to-one.

Each end of an association has an optional role name. These role names help focus the purpose of the association between the connected classes. Role names are necessary for associations connecting a class to itself. Role names on associations connected to a given class must be unique. An example of role names is HUSBAND and WIFE on the association between two PERSON objects.

Associations can possess attributes. These are called link attributes. If associations need behavior characteristics, then they can be modeled as classes. As an example, a FOOTBALL TEAM has a SCHEDULE to play other FOOTBALL TEAMs. SCHEDULE could be modeled as a link with the attributes GAME TIME and SCORE. If needed, the RESCHEDULE operation could be added to the association.

A special form of association is aggregation. This is a "part-of" or "consists-of" relationship. The *forward* end is the *assembler* of all the *inverse* ends or *components* it connects to. An example of aggregation is a BOOK that is composed of a COVER and PAGES.

Like classes and attributes, associations can be derived from other derived or base associations. A derivation string is used to describe how and where the association originated. For example, the aggregation of WASTE BASKET parts might be derived from a variety of other associations flagged for deletion.

Certain associations may need to be restricted or constrained by other associations. An association that is a subset of another association is a good example. The SONS association between HUSBAND and WIFE is a subset of the CHILDREN association.

Operations in one class can propagate across an association to another class. The propagation message is the operation itself with its name and signature. Propagated operations are typically used in aggregate structures to cause chain reactions. For example, when a FLAG object draws itself, it propagates the DRAW operation to its STARS and STRIPES components.

3.3.1.3 Generalization and Inheritance Data Requirements Generalization and inheritance (or specialization) are inverses of each other. Generalization extracts the common attributes and/or operations of a group of classes and places them in a higher level superclass. The lower level subclasses inherit all attributes and operations from the superclass.

The OMT allows single or multiple inheritance. Single inheritance means a subclass inherits from a single superclass. Multiple inheritance allows a subclass to inherit from more than one superclass. Overlapping inheritance occurs in multiple inheritance cases where subclasses of a superclass have non-disjoint subclasses of their own (17:65).

An inheritance relationship may have an optional discriminator. The discriminator "is an attribute of enumeration type that indicates which property of an object is being abstracted by

a particular generalization relationship (17:39)." For example, a CAR may have the subclasses FOREIGN CAR and AMERICAN CAR. PATRIOTIC could be used as the discriminator.

3.3.1.4 Grouping Constructs Data Requirements The object model consists of class diagrams and object diagrams (17:23). Class diagrams contain classes, associations, and generalizations. Object diagrams contain instantiated objects and links. Since large, complex systems are easier to understand when broken into smaller pieces, class and object diagrams must allow logical and physical decomposition into smaller modules and sheets.

A module is a logical or conceptual view of a group of object model elements. Each module has a name. Sheets are physical groupings of object model elements. Each sheet has a title and page number. It must fit on a single printed page. Modules are composed of one or more sheets. Object model elements can be duplicated in other modules or sheets.

3.3.2 Dynamic Model Data Requirements The dynamic model is used to specify when and how changes occur inside an object. It describes the *control* of the objects in the system–the sequence of operations that occur in response to stimuli. The dynamic model uses a state-transition methodology to capture the controlling aspects of the system. The fundamental components of state-transition diagrams are states and transitions. Other elements are superstates, control splits, and control synchronizations. Furthermore, state-transition diagrams can be nested inside other state-transition diagrams.

A state represents an abstraction of attribute values of an object. For example, the state DARK might exist for a ROOM object to describe the state when the LIGHT attribute is off and the TIME ranges from 6pm to 6am. States have a name and can be either an initial, intermediate, or final state. The initial state shows the starting location for a state diagram. The final states show where the object control terminates. Intermediate states show the various stages of the object's

life-cycle. Each state-transition diagram has one initial state, one or more intermediate states, and zero or more final states.

States can activate or control an optional activity, and they can perform zero or more internal actions. Activities are used to describe behavior that executes for a duration of time. They have names and code attributes to describe this behavior. Like activities, actions also have names and code, but actions are used to describe instantaneous behavior. Internal actions are performed in one of three ways: when a state is first entered, when it is exited, or when an event (described later) needs to be handled within the state. Both activities and actions can generate events. Examples of activities and actions for a CLOCK object are the SOUND ALARM activity and RESET ALARM action.

Transitions in a state diagram provide the mechanism to move from one state to the next. Any combination of the following items can be attached to a transition: external actions, guard conditions, and events. Transitions without any of these items can occur and are triggered automatically when the "from" state completes any activities and internal actions. External actions are similar to internal actions, except they occur during transitions. Guard conditions are Boolean expressions used to describe what conditions are needed before a transition can "fire". For example, a transition to the PAID BILL state occurs only if the CHECK IS SIGNED attribute is true. Finally, events can cause transitions, and they can be sent to other states or classes.

Events are the "driving force" behind the state transition diagram. They provide the external stimuli and messages mentioned above. Events have names and may also have event attributes. These attributes are parameters or messages that are conveyed along with the event to its recipient. Examples of events are:

- **HONK event sent to SUNDAY DRIVER**
- **TURN (WHITE) event sent to CHESS BOARD**
- **KISS (LIPS, CHEEK, NOSE) event sent to FAMILY (WIFE, KIDS, DOG)**

Furthermore, events can be categorized and grouped into an event hierarchy. This is purely optional. It provides a convenient way to organize and relate similar states. Sub-events inherit characteristics from super-events. For example, EAT and DRINK may be sub-events of CONSUME.

Superstates are recursive elements used in state-transition diagrams to provide state generalization and concurrency. Each superstate has a name and is composed of state subdiagrams or substates–which in turn can be composed of lower-level superstates. The MASTER SWITCH ON superstate in a computer object is an example of a superstate that contains the substates CPU ON, MONITOR ON, and PRINTER ON. Also, an example of concurrent superstates is EATING and TALKING substates for a RUDE superstate.

Normally, transitions flow from one state or superstate to another. It's sometimes necessary to fork or split control into two or more threads, then rejoin and synchronize some or all of those threads later. Control splits and synchronizations provide this capability. Splits are junction points with one incoming transition and two or more outgoing transitions, and synchronizations are junction points with one outgoing transition and two or more incoming transitions. For example, at 8am, the FAMILY object in the TOGETHER state transitions and splits to the WORK, SCHOOL, and SHOPPING superstate threads. At 5pm, these threads synchronize back to the TOGETHER state.

Transitions connect the "node" elements of the state diagram, namely, states, superstates, splits, and synchronizations. Any binary combination of these nodes can be connected, that is, a state can transition to a superstate, split, synchronization, or another state, etc.

3.3.3 Functional Model Data Requirements The functional model describes the data processes or transformations that occur within the system under study. It shows where the data flows through the system, that is, which classes, attributes, and operations are the targets and sources of the information flows. The functional model follows the methodology of data flow diagrams.

The main elements or nodes of the functional model are processes, data flows, data stores, and actors. The lesser elements of the functional model are control flows, store flows, and data flow junctions. Each of these elements is discussed in this section.

A process transforms data flows. Processes should have at least one input and one output data flow. Each process has either a descriptive name or narrative text, or both. Processes describe some of the behavioral aspects of operations in the object model classes. SORT is an example of a process that inputs a LIST data flow and outputs a SORTED LIST data flow.

Processes can be either atomic or complex. A complex process expands into a data flow diagram sublevel. An atomic process is the lowest level process, and it can't be further expanded. Processes that were considered atomic during the analysis phase of development may actually be complex in the design and implementation phases.

Data flows are intermediate data values passed between connecting nodes. These flows have names describing their data values. Data flows connecting to external sources/targets such as actors and data stores are inputs/outputs and sources/sinks respectively. The names of these data flows are the attributes or class names of the external sources/targets they connect. Data flows with external class names can be omitted because their name is implied.

A data store is an object that passively stores and provides data. They have a name and an optional file name. The file name is used to describe the actual physical storage location. Data flows from data stores are inputs, and data flows to data stores are outputs. ADDRESS FILE is an example of a data store.

Actor objects in the functional model are explicit objects in the object model. They are active entities that produce (source) and/or consume (sink) data. Their behavior should not appear in the functional model. Like classes, actors have a name and all the other characteristics and components associated with classes. For example, a CLOCK is an actor that produces a TIME data flow.

A control flow is a decision path from one process to another. It is a Boolean expression that regulates control flow from one process to another. This information should be duplicated in the dynamic model and is used sparingly in the functional model to help clarify process flow. An example of a control flow is SOLDIER IS OVERWEIGHT between the two processes CHECK WEIGHT and ENROLL IN FAT-BOY PROGRAM.

A flow from a process that results in a data store is called a store flow. Typically, these data stores are temporary storage for intermediate data transformations. For example, in a batch processing environment, a payroll system may have a CALCULATE PAY process that extracts HOURS WORKED from the time card and PAY RATE from the employee file. A temporary data store called CHECK REGISTER may be generated for later use by a PRINT CHECKS process.

Data flow junctions are convenient alternatives to requiring each flow to connect to only two nodes. Data flow junctions are duplications, compositions, and decompositions. Duplications allow one data flow to split into two or more flows with identical copies of the data going to each new flow. Compositions merge two or more data flows into one composite flow. For example, ADDRESS is a merged composite of NAME, STREET, CITY, STATE, and ZIP. Decomposition splits composite flows into two or more parts.

3.3.4 Model Cross-Links Data Requirements The above sections mentioned some of the ways all three OMT sub-models are related. This section summarizes all these cross-links.

The data model is composed of object diagrams from the object model, state transition diagrams from the dynamic model, and data flow diagrams from the functional model. The data model has a name which is typically the name of the system being modeled. For example, the ELEVATOR PROBLEM data model is composed of object, state-transition, and data flow diagrams describing the objects, behavior, and processes involved in the problem space.

Every class in the object model has an optional state diagram to describe its internal behavior. Actors and data stores in the functional model are both classes. Data flows can be either a class or an argument attribute.

Class operations have behavior that can be described by activities, actions, and events in the dynamic model. Data processing information can be described by processes in the functional model.

Finally, attributes in classes are constrained according to the state the class is in. For example, a PERSON object has a HEART RATE attribute. While in the SLEEP state, HEART RATE is constrained to a much lower rate than during the EXERCISE state.

3.4 Drawing Model Requirements

The drawing model must capture all the graphical notations of the OMT. These elements should not be dependent on a specific graphics software or hardware platform. This model must not duplicate information found in the data model (see the next section on binding tool requirements to see how the two models are linked). For each of the OMT data elements, Rumbaugh provides a corresponding graphical or drawing element. Appendix A contains diagrams showing the notations of the OMT.

The drawing model must provide the capability to draw any of the OMT notations. Position and size are required for all drawing elements. Drawable elements can be either text, nodes, arcs, or any combination of the three. Text should allow style (normal, boldface, italics) and alignment (right, center, left). Nodes are of various shapes as needed to depict the OMT states, classes, process, etc. Arcs can be straight lines or curves, but poly-lines or poly-curves (arcs with multiple vertices) are needed to allow complex routing. Arcs must also allow various drawable shapes to be placed at the arc ends such as arrowheads, circles, diamonds, and text.

The aggregation relationship and generalization relationship can be represented in one of two forms as shown in Appendix A. These relationship arcs can either be drawn from one class to one other class, or they can be "yoked" from the superclass or assembler to all its subclasses or components respectively. This yoke is a horizontal bar that provides a junction to route the superclass or assembler to its subclasses or components.

3.5 Binding Tool Requirements

While presenting his OMT methodology, Rumbaugh simultaneously discusses the informational (or data) and notational (or drawing) elements of the various pieces of the OMT. For each data element of the OMT, there is a corresponding drawing element. Elements in the data model correspond one-to-one to elements in the drawing model. However, this relationship is unidirectional. Many elements in the drawing model cannot be mapped directly back to the data model. For example, an arc can be a relationship, data flow, transition, etc. Also, a "yoke" (described above) isn't represented in the data model. Therefore, although the drawing model can adequately capture all the essential information of an object-oriented system, an intelligent or "binding tool" is needed make sense of all the graphical figures. The purpose of the binding tool is to bind the drawing information to the data information.

Rather than duplicate and embed information from the data model into the drawing model, the binding tool should provide the necessary "hooks" into the data model to allow the drawing model to access the necessary data. This helps maintain data integrity. For example, rather than duplicate the actual text needed in the drawing model to put a class name in a class rectangle, the drawing tool should go to the binding tool to request the necessary text from the state data object.

The binding tool should also provide a basic "front-end" to create and populate the data and drawing models. A dispatching method is needed in the binding tool to route user queries, commands, and messages to the appropriate classes and objects. The binding tool must also provide

any additional object management and coordination features not found in the individual classes and objects.

3.6 Requirements Summary

These requirements were used to drive the next phases of system design and implementation. While the thesis advisor and sponsor provided certain requirements input such as system design methodology and implementation language, the OMT data and drawing requirements and specifications were derived solely from Rumbaugh's text (17). The next chapter explains the data and drawing model system designs.

IV. System Design

This chapter explains the conceptual view and designs of the system that were developed to meet the thesis requirements. Many decisions were made during the system design process. Numerous changes and enhancements were made to the original designs. The various sections of this chapter elaborate on some of these design decisions and revisions. However, only the final versions of the design diagrams are actually shown. A complete collection of all the design diagrams is found in the appendices.

The conceptual view is presented first to show how the entire system interfaces to external entities. Next, the basic or core class that is inherited by all other classes is described, followed by an explanation of the data and drawing models. Afterwards, the binding tool / driver program is discussed. Finally, this chapter concludes with a summary of the entire design process and explains the steps taken to prepare the designs for implementation.

4.1 System Conceptual View Design

At the highest level, the OMT system consists of the data and drawing models, the object database or repository, the object manager, a graphics library, and a driver program. Figure 7 shows these entities. The boundary of the system is drawn in the diagram to encapsulate the system and show where interfaces are needed to communicate with the system. The internal characterics of these high level system components are described in detail in later sections of this chapter. This section discusses how these components interface at the top level.

The data model provides the interface to the data objects in the object repository. All accesses to the data objects are routed through this module. The drawing model provides the interface to the drawing objects in the repository. All accesses to the drawing objects are routed through the drawing module. When the drawing model needs data information (i.e., the text in a class rectangle), it derives this information by interfacing through the data model.

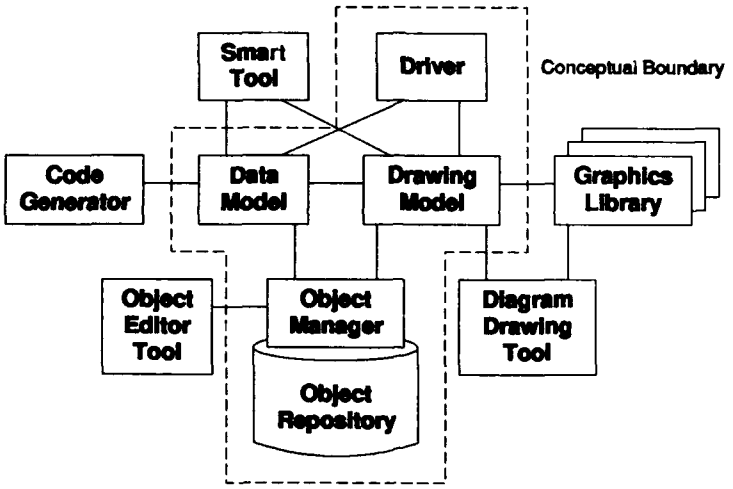


Figure 7. System Conceptual View

A graphics library provides the concrete operations needed by the drawing tool to draw the created OMT models on the targeted platform. The actual implementations of these graphic functions are environment-specific (i.e., MOTIF, OpenLook), but the interfaces are standardized for all environments. This eliminates the need to rewrite the drawing model when porting it to a new graphical environment. At a minimum, this library must support the functions needed to draw all the basic graphic elements shown in Figure 36 (described in later sections).

Standard Ada lacks many of the object management functions needed for this tool. The object management module was created to fill this need. Many of the features of this module were not know until implementation. The implementation features of the manager are described in the implementation chapter. Some of the known design features are:

- ***Allocation and deallocation of object handles*** - these functions provide the uniqueness needed for object identity.
- ***Object list management*** - this provides the structures and functions needed to organize and access the objects in each class.
- ***Object master list*** - this consolidates the individual object lists from each class into the central repository.

The driver program was designed to satisfy the requirement to create and populate the object repository. It interfaces to both the data and drawing models as shown. Its main function is to accept user responses and dispatch messages to and from the data and drawing models. Its purpose and function is described in greater detail in the implementation chapter.

Outside the boundary of the system, various entities exist that may want to use and communicate with the system. Figure 7 shows some of these possible entities. A object editor tool could be used to query and update the objects in the repository directly, rather than go through the models. This could be used in place of the simple driver written for this thesis. A smart tool could be used to do consistency checking or to create drawing models from existing data models. This could be used to verify design correctness or to generate visually optimized drawings. A code generator could be used to build executable code from the object models in the repository. Finally, and probably most importantly, a graphical drawing tool could be used to visually create and populate the OMT models.

4.2 Basic Class Design

In the early phases of design, the individual classes were created with all their attributes and methods located locally. As more and more classes emerged, it became apparent that all classes shared some common characteristics. Certain attributes and operations were showing up in every class. At that point, an important design decision was made. Rather than duplicate these attributes and methods in every class, a BASIC core class was designed to encapsulate this commonality into one superclass. Figure 8 shows this class. The principles of generalization were used to create this top-level superclass (17:38–43). Every class object in the entire system inherits the attributes and behavior of the BASIC class.

In order to meet the requirement that every instantiated object must be uniquely identifiable, an attribute was needed to hold a unique object identifier. This identifier is the HANDLE attribute

Figure 8. Basic Class Design

in the BASIC class. Uniqueness is provided by the object management module described above. The HANDLE attribute is called a *candidate key*. Rumbaugh discusses candidate keys in his text (17:71–72). "A *candidate key* is a minimal set of attributes that uniquely identifies an object or link." This attribute is viewed externally as a read-only attribute. Only the object manager can set this value during object creation or deletion.

The methods in the BASIC class can be categorized into three areas: access, instantiation, and persistance operations. Access operations get, set, and list attribute values. Instantiation operations clear, create, and delete instances of class objects. Persistance operations load and save objects from and to secondary storage. The methods in each category are described below, but before discussing these methods, the concept of *class operations* should be understood. Any operation that acts on the class itself is a class operation (17:71). LIST, CLEAR, CREATE, LOAD, and SAVE all fit this description, and therefore, they are class operations. (The names of class operations are preceded by a dollar sign.) This will become apparent after reading the descriptions of these operations.

- CLEAR - clears or deletes all instances of objects in the given class
- CREATE - creates an object instance in the given class
- DELETE - deletes an instantiated object
- GET - gets the attributes of an object
- SET - sets the attributes of an object
- LIST - list all objects in the given class

- LOAD - loads all the objects for the class
- SAVE - saves all the objects for the class

4.3 Data Model Design

The data model design is shown in Appendix C. The data model class has a name. The NAME attribute is used to give a project name or title to the entire data model that is being created. The components of the data model are the three sub-models of the OMT. The three sub-models of the OMT are described with object, state, and data flow diagrams. The data element designs of each of these sub-models are discussed in detail in Appendix B. This section briefly describes the overall design of the data model.

4.3.1 Object Model Data Design The object model uses an entity-relationship methodology called object diagrams. Appendix D contains all the data model diagrams for the object model. The complete design of the object diagram is broken into nine modules. These modules are:

- *Class Design Module* - models the data elements associated with classes such as class names, attributes, operations, attribute domains, and behavior. Behavior was added to provide an alternate way to describe behavior in the object model rather than in the dynamic and functional models.
- *Inheritance Design Module* - models the data elements associated with inheritance such as overlapping and disjoint inheritance, discriminator attributes, and connections between superclasses and subclasses.
- *Association Design Module* - models the data elements associated with associations such as aggregations, binary and ternary associations, connections between associations, and associations modeled as link attributes or classes.
- *Class Connection Design Module* - models the data elements associated with connections between classes such as the connected classes, class roles, multiplicity, and qualifiers.

- *Derives Design Module* - models the data elements associated with derived types such as derived classes, derived associations, derived attributes, and the text strings describing where and how these types are derived.
- *Constrains Association, Propagates Operation, and Operation Types Design Module* - models the data elements associated with these various concepts, namely, constraints between associations, propagation of operations between classes, and types of operations.
- *Constrains Class, Attribute Types, and Class Types Design Module* - models the data elements associated with these various concepts, namely, class constraints, and types of attributes and classes.
- *Instantiates Design Module* - models the data elements associated with instantiation of objects such as objects, object identification, links, and attribute values.
- *Object Diagram Design Module* - models the data elements associated with object diagrams such as class diagrams, instance diagrams, and grouping constructs (modules and sheets).

4.3.2 Dynamic Model Data Design The dynamical model uses a state-transition methodology. Appendix E contains all the data model diagrams for the dynamic model. The complete design of the state-transition diagram is broken into four modules. These modules are:

- *State Design Module* - models the data elements associated with states such as state types (initial, intermediate, final), activities, and internal controlling actions (entry actions, exit actions, and internal event handlers).
- *Transition Design Module* - models the data elements associated with transitions such as external actions, guard conditions, and events.
- *Transition Connection Module* - models the data elements associated with connecting transitions to states, superstates, and control splits and synchronizations.

- *State Diagram Module* - models the data elements associated with state diagrams such as combining state-transition elements into a diagram, superstate types, event generalization, and activity expansion.

4.3.3 Functional Model Data Design The functional model uses a data flow methodology. Appendix F contains all the data model diagrams for the functional model. The complete design of the data flow diagram is broken into three modules. These modules are:

- *Flow Connection Modules* - these two modules model the data elements associated with creating data flow objects such as actors, processes, data stores, data flows, control flows, and store flows. The first module models the data elements associated with connecting these various objects together. The second module models the data elements associated with duplicating, composing, and decomposing data flows.
- *Data Flow Diagram Module* - models the data elements associated with building data flow diagrams (DFD) such as grouping the DFD objects into a specific diagram, and creating nested levels of DFDs.

4.3.4 Model Cross-Links Data Design The OMT sub-models all have certain elements that cross-link and relate to the other sub-models. Appendix G contains all the model design digrams for these cross-links. These design diagrams consist of four modules. These modules are:

- *Class Cross-links Module* - Figure 32 shows the cross-links to the class object. Every class has its own state diagram describing its controlling behavior. This forms a relation between the object and dynamic models. Furthermore, an actor is a class object. Data stores and data flows can also be class objects. This forms a relation between the object and functional models.
- *Operation Cross-links Module* - Figure 33 shows the cross-links to the operation class. Activities, actions, and events are all operations in the class module. This forms a relation between

the object and dynamic modules. Also, processes in the functional model correspond to zero or more operations in the object model.

- *Data Flow Cross-links Module* - Figure 34 shows additional cross-links for data flows. If a data flow is not a class object as mentioned above, it can then be either an attribute or an argument.
- *State Cross-links Module* - Figure 35 shows cross-links to the state class. The attributes of a class are constrained according to the state the class is in. The CONSTRAINS association has a constraint link-attribute. This cross-link further relates the object and dynamic models.

4.4 Drawing Model Design

Figure 37 in Appendix I shows the top-level drawing components of the OMT. The drawing model group is composed of optional data model text and zero or more object, state, and data flow diagram groups. The data model text has a name that is derived from the data model class in the data model. This following subsections first discuss the general graphical classes of the drawing model, then describe each of the three diagram groups in the drawing model group. Appendix B contains a detailed description of the three diagram groups.

The associations in all the drawing model diagrams turned out to be aggregations. This resulted because all the OMT drawing elements are assembled from lower-level graphical components. Furthermore, for every class and association in the data models, there is a corresponding class and aggregation in the drawing models. This resulted because for every data element of the OMT that Rumbaugh discusses in his book, he also discusses its corresponding drawing element.

The derivations for all derived attributes, classes, and associations in the drawing model diagrams are either implicitly or explicitly stated. If not explicitly stated, the derivation is obvious. For example, DATA-MODEL-TEXT.NAME is obviously derived from DATA-MODEL.NAME in the data model, or the components of DATA-MODEL-GROUP are obviously the same as the

components of DATA-MODEL in the data model. Explicit derivations are shown when it is less obvious where the derived originated from. For example, the CONTROL GROUP components in Figure 47 are derived from the listed associations in the dynamic model data design. All derived types in the drawing model are discussed individually in the following subsections.

4.4.1 Graphical Classes Drawing Design Figure 36 in Appendix H shows all the graphical elements of the OMT notation. Every class in each of the drawing model designs has a corresponding graphical element in this figure. All the graphical elements were abstracted from the OMT notations found in the diagrams in Appendix A.

The top-level class in the drawing model is the DRAWABLE superclass. Every item that can be drawn is a subclass of this superclass. A drawable has an abstract DRAW operation. Each graphical subclass provides a concrete draw operation that does the actual work of drawing itself.

The GROUP class is a container class for grouping drawable objects. A group consists of an anchor and zero or more drawables. The anchor is used as a common reference point to which all drawables in the group anchor themselves. The concept of anchors was taken from the Object Maker modeling tool (18). Object Maker uses anchors to bind one graphical object to another. This allows groups of objects to move in reference to the anchor point. For example, when a node moves, all objects attached to its anchor also move, and any arcs anchored to one end of the node are stretched or shortened accordingly (rubber band effect).

A drawable can be either a node, arc, or text. Text drawables have position, size (point size), font, style, and align attributes. Arcs have an ordered array of two or more points and a style (solid or dashed line) attribute. An arc is composed of zero, one, or two tip groups. These tip groups are anchors on the arc ends, and they are used to place arrowheads, text, etc., on the arc tips.

Nodes are two-dimensional icons that come in a variety of shapes. A node has a position attribute. Junction nodes provide splitting and merging capabilities for multiple arcs. Junctions have two subclasses: points and yokes. Point junctions allow arcs to split or merge at a single

point, and yoke junctions allow arcs to split or merge at a horizontal bar. Yokes add length and direction attributes to its inherited attribute list (position).

Polygon nodes have an ordered array of three or more vertex points and a fill pattern (solid, empty). Rectangles are a subclass of polygons and are constrained to have only four vertices. Two of the vertices are the upper-right and lower-left corners, and the other two are derived from the first two points.

Conic nodes are open or closed eliptical arcs or pies. They have length, width, start and finish angles, and a fill pattern. The start and finish angles follow the polar orientation used in trigonometry (zero degrees is at the three o'clock position, and angles are measured counter-clockwise). Conics are specialized into elipses which are closed conics, that is, the start and finish angle are 360 degrees apart. Elipses are specialized into circles which are constrained to equal lengths and widths. A rounded rectangle inherits features from both rectangles and circles. The radius of the rounded corners are described by the circle superclass, and the size is described by the rectangle superclass.

4.4.2 Object Model Drawing Design Appendix J contains all the drawing model diagrams for the object model. The object model drawing design consists of nine modules. These modules are:

- *Class Group Module* - models the drawing elements associated with classes such as class rectangles, class name text, attribute list, operation list, and class constraints.
- *Inheritance Group Module* - models the drawing elements associated with inheritance such as inheritance arcs, discriminator text, inheritance yokes (alternate notation for inheritance arcs), and inheritance notations for disjoint or overlapping inheritance.
- *Association Group Module* - models the drawing elements associated with associations such as association arcs and name text, associative classes or link attributes, and aggregation groups.

- *Connection Group Module* - models the drawing elements associated with connecting or relating classes such as class role text, multiplicity symbols and text, and qualifier text.
- *Derivation Group Module* - models the drawing elements associated with derived types such as the derivation arc or slash symbol, and the derivation text.
- *Constraint Group Module* - models the drawing elements associated with constraints on classes and associations such as the constraint text and the constraint arc between constrained associations.
- *Propagates Group Module* - models the drawing elements associated with propagated operations such as the directed arrow and the operation text.
- *Instantiates Group Module* - models the drawing elements associated with instantiating objects such as object rounded rectangles, link arcs, and relationship arcs between objects and the classes from which they are instantiated.
- *Object Diagram Group Module* - models the drawing elements associated with building object diagrams such as class and instance diagrams groups, and grouping constructs (module and sheet groups).

4.4.3 Dynamic Model Drawing Design Appendix K contains all the drawing model diagrams for the dynamic model. The dynamic model drawing design consists of four modules. These modules are:

- *State Group Module* - models the drawing elements associated with states such as state nodes, state text and text for the control group.
- *Transition Group Module* - models the drawing elements associated with transitions such as transition arcs, guard condition text, event text, and arcs for sending events to other classes.
- *Superstate Group Module* - models the drawing elements associated with superstates such as rounded rectangles to hold all the substates and the superstate name text.

- *State Diagram Group Module* - models the drawing elements associated with building state diagrams such as the rectangle to hold the state diagram drawing objects.

4.4.4 Functional Model Drawing Design Appendix L contains all the drawing model diagrams for the functional model. The functional model drawing design consists of four modules. These modules are:

- *Node Groups Module* - models the drawing elements of all the nodes in the functional model. These node groups are:

  - Process Group
  - Data Store Group
  - Actor Group

- *Arc Groups Module* - models the drawing elements of all the arcs or flows in the funcional model. These flow groups are:

  - Data Flow Group
  - Control Flow Group
  - Store Flow Arc (shown in the flow connection group)

- *Flow Connection Groups Module* - models the drawing elements of the flow connection groups. The purpose of the connection tuples is to pair-up the components in the respective group. The three groups in this module are:

  - Data Flow Connection Group
  - Store Flow Connection Group
  - Control Flow Connection Group

- *Data Flow Diagram Group Module* - models the drawing elements associated with creating data flow diagrams such as diagram name text and diagram leveling.

4.5 Binding Tool Design

The derived types (classes, associations, and attributes) in the drawing model are all binding elements. Many of the binding tool requirements were placed in the drawing model because it provided a more direct route than going through an intermediate tool. Derived types are natural concepts in the OMT methodology for binding elements to other elements (17:75–76). The actual derivation process of these derived types was deferred until implementation.

Another binding requirement is implicit in the drawing model designs. As mentioned previously, for every design element (classes and associations) in the data model, there is a corresponding equivalent design element in the drawing model. This one-to-one relationship between the two models is implied in the similarities of the class and association names, as well as the general structural similarities between the two model design diagrams.

4.6 Design Summary

One important issue that came up during the design was whether to model relationships as aggregations or associations. Rumbaugh discusses this issue in his text (17:58). "The decision to use aggregation is a matter of judgement and is often arbitrary. Often it is not obvious if an association should be modeled as an aggregation." The rule used to determine whether to use aggregation or association was if two objects are independent, then relate them using association. If there is a whole-part relationship, then use aggregation. In the drawing model, this distinction was easy to make because all the graphical elements assembled to form the various icons and notations of the OMT, and therefore, were modeled as aggregates. However, in the data model, this distinction was less obvious. Aggregation was only used in the data model when data elements could be considered lesser pieces of a higher data structure. Sometimes an aggregation in the drawing turned out to be an association in the data model. For example, in the drawing model, an activity's text is part of a state's icon, but in the data model, activities and states are independent objects.

Another issue considered during design was whether to allow inconsistencies and incompleteness to exist when creating object, state, and data flow diagrams. If inconsistencies were not allowed, then the data and drawing models would need to be smart enough to catch such cases. This additional overhead could be enormous. Furthermore, incomplete designs could not be created, saved, and filled in later. This would disallow the gradual construction of designs. Therefore, inconsistencies and incompleteness were allowed in the designs.

In preparing for implementation, all generalizations were reviewed and "flattened" to single level inheritances. This was done because of the limitations of Ada to provide adequate inheritance features. However, the inheritance hierarchy in the graphical class design was not changed because it would have resulted in a very unnatural design.

V. *Implementation*

This chapter discusses the implementation aspects of the thesis research. The implementation of the OMT data and drawing models was the most intense phase of the research effort. Because of the inabilities of standard Ada to fully support all the features of an object-oriented system, various design decisions were made early on and throughout the implementation phase to overcome Ada's shortcomings. The beginning sections of this chapter explain these decisions. The later sections describe the actual implementation of the data and drawing models, the message dispatcher, and the simple driver that was built to create and populate the object repository. The chapter concludes with a summary of the implementation phase.

5.1 Using Ada to Implement an Object-Oriented Design

The system designs presented in the previous chapters were implemented using a standard Ada compiler, Verdix Ada Version 6. The translation of the object-oriented designs to object-based Ada introduced many challenges. A variety of acticles have addressed the question of whether Ada is really an object-oriented programming language (OOPL) (27, 9, 3). Most of these articles agree that although Ada falls short of a true OOPL, there are some fairly good "work-arounds" to help simulate certain OOPL features. This section describes the "work-arounds" that were used for this research effort by showing how class structures and inheritance were implemented. A later section in this chapter discusses another "work-around" used to overcome Ada's lack of run-time class/method resolution.

5.1.1 Class Structure Implementation The basic element of an object-oriented system is the class entity. The basic ingredients of a class are identity, state, and behavior. These items are encapsulated inside the class entity, and access to them is restricted. Ada provides the package unit to accomplish almost everything described above. The package name provides the class identity, private data type declarations provide the class state (attributes), and package subroutines provide

the behavior. The package encapsulates these items into a unit where access is limited to visible subroutine interfaces. Therefore, all classes for the designs in this thesis were implemented using the Ada package unit as suggested by various object technology experts (12:92–97) (17:360) (27:20–21).

The basic structure of a class using Ada packages is shown below. The class attributes are declared in a private data block. Access to the internal structure of variables that are declared of private type is strictly prohibited. Appropriate GET and SET procedures provide the access methods needed to query and change class attributes. Every class should also have a constructor and destructor operation to CREATE and DELETE instantiations of objects (17:346–347).

```
-------------------------------------------------

   package Class is

      type Attributes_Type is private;

      procedure Create  (...);
      procedure Delete  (...);
      procedure Get     (...);
      procedure Set     (...);
      procedure Do_This (...);
      procedure Do_That (...);
      ...

   private

      type Attributes_Type is record
         Attribute_1 : ... ;
         Attribute_2 : ... ;
         Attribute_3 : ... ;
         ...
      end record;

   end Class;

-------------------------------------------------
```

At the beginning of the implementation phase, it was hoped that all classes would be patterned after this ideal class structure. However, some of the compiler dependency limitations in Ada prevented this strict adherence. Deviations from this ideal class structure are described in later sections of this chapter.

5.1.2 Inheritance Implementation Inheritance is another object-oriented feature absent from standard Ada. Rumbaugh suggests three ways to deal with inheritance in a design when inheritance is unavailable in the implementation language (17:347–348):

1. Avoid it
2. Flatten the class hierarchy
3. Break out separate objects

All three methods were applied to the system design models before implementation began. Inheritance was avoided whenever possible during the design phase. When it wasn't possible to avoid, then the class hierarchy was flattened, and all subclass attributes were moved up to the superclass. For example, referring to Figure 28, the SUPERSTATE class originally had two subclasses, concurrent and generalized superstates. The concurrent superstate was further broken into asynchronous and synchronous concurrent superstates. To flatten the hierarchy, the synchronous/asynchronous subclasses were moved up into their superclass. Finally, some class hierarchy designs were left as is, but the subclasses and superclasses were all implemented as separate objects. For example, referring to Figures 32 and 33, all the subclasses of the CLASS and OPERATION superclasses were implemented as separate objects.

Avoiding inheritance hierarchies or breaking them down into separate objects required no further implementation concerns. However, flattened hierarchies that couldn't be broken out required special consideration. Three possible solutions are presented and discussed below.

Rumbaugh suggests that single level inheritance can be implemented in Ada by using variant records (17:349–350). While this method allows attributes to be inherited, operations cannot be inherited. Furthermore, inefficiencies may arise because the storage space for each object in the hierarchy is statically set to the largest object. When some objects are significantly smaller than others, space is wasted. Therefore, variant records were not used to implement flattened inheritance.

The second way to deal with flattened inheritance involved adding a discriminator attribute called SUBCLASS to the superclass. In this way the superclass and subclasses all reside in the same implemented class, and only a discriminator attribute distinguishes them. Rumbaugh explains a discriminator as an attribute "that indicates which property of an object [or class] is being abstracted by a particular generalization relationship (17:39)." For example, in Figure 25 the STATE specialization uses the SUBCLASS discriminator. For all flattened hierarchies where the subclass' characteristics of attributes, operations, and associations could be combined into the superclass, the SUBCLASS discriminator worked well. However, a third way was needed to implement inheritance when subclass characteristics could not be combined into the superclass.

Meyer poses and answers the following question in his article (3): "Can inheritance be simulated with genericity?" He goes through a one-page explanation of why he feels genericity fails to do the job. However, the author gets side-tracked. Like Rumbaugh, he suggests the use of variant records. Meyer then goes on to discredit this method because it fails to allow extendibility, which is the major purpose for inheritance. The variant record structure would have to be amended each time new subclasses are added to the hierarchy. But what about generics? Meyer never returns to the genericity vs. inheritance question!

Jones asks the same question about inheritance and genericity is his own article (9). He provides concrete examples of how well Ada generics can implement inheritance, both single and multi-level hierarchies. He even shows an example of using generics to implement multiple inheritance. This author adopted Jones' ideas to implement inheritance. Since the hierarchies had already been flattened, this task was much simpler. The implementation of the basic class (described later in this chapter) used a generic package to create a top-level superclass for all other classes in the data and drawing models.

5.2 Object Manager Implementation

In a true object-oriented programming environment, many object management features are automatically provided by the language. Other features can be easily added to the manager. Because there are no class or object data types provided in standard Ada, object management does not exist. Therefore, to manage the simulated class types described earlier in this chapter, a simple object management system was developed. After evaluating the programming environment for this thesis, it was determined that the object manager needed to provide the following:

1. Central location for common object types
2. Assurance of unique object identities
3. Storage for object repository

The following subsections describe the basic responsibilites of the object manager and show how these responsibilities were implemented. Appendix M.3 contains the Ada code for the object package. Object management is performed in this package.

5.2.1 Common Object Types A few common types were needed by either all classes or a certain category of classes. These types were extracted from the individual classes and placed in a common types "pool" inside the object manager package. It is good programming practice to co-locate shared types and constants. If these declarations need to be changed in the future, changes occur in one place and affect all uses of these types and constants. The following listing shows these common declarations taken from Appendix M.3.

```
------------------------------------------------------------

   subtype handle_type is integer;

   null_handle : constant handle_type := 0;

   package handle_list_pkg is new sequential_list_pkg (
      item_type => handle_type,
      "<"       => standard."<");

   type aggregation_type is (
      assembler_object,
      component_object);

   type aggregation_objects_type is
      array (aggregation_type) of handle_type;

   type binary_association_type is (
      forward_object,
      inverse_object);

   type binary_association_objects_type is
      array (binary_association_type) of handle_type;

   type ternary_association_type is (
      forward_object,
      middle_object,
      inverse_object);

   type ternary_association_objects_type is
      array (ternary_association_type) of handle_type;

   handle_search_error : exception;

------------------------------------------------------------
```

Each class has a **handle** attribute used to store the value of the unique object identifier (described in the next subsection). Therefore, the type declaration for this attribute was placed in the common pool. A **null_handle** constant was added to provide an initialized handle value. It was also anticipated that a list of handles may be needed. The **handle_list_pkg** serves this purpose. This list was instantiated from a generic sequential list package (Appendix M.1).

The **handle_search_error** exception is raised by all classes when a non-existant object is searched for. For example, an attempt to delete a non-existant object will cause this error to be raised. There is a commentary in the summary of this chapter on the use of exceptions in an object-oriented system.

The last category of common types is used in associative classes (associations modeled as classes). Associative classes are described later in this chapter. Figure 18 shows the various roles of the connections attached to associations. The common pool declares these roles as enumerated types (**aggregation_association_type**, **binary_association_type**, and **ternary_association_type**). It also declares the array types needed to store the links that are instantiated from these associative classes (**aggregation_association_objects_type**, **binary_association_objects_type**, and **ternary_association_objects_type**).

5.2.2 Object Uniqueness One of the requirements of an object is that it must be uniquely identifiable. The object manager provides the facility needed to satisfy this requirement. The original plan to implement object identity involved generating a free-list of unique object handles. Whenever an object was created or deleted, a handle was taken from or put back into the free-list. However, this plan was rejected for the following reasons:

1. *Data integrity* - Data integrity must be satisified in order for uniqueness to exist. Semaphores, tasking, crash recovery, and other features would be needed to implement data integrity. These are all complex and difficult items to implement.
2. *Storage space* - A full free-list must be large enough hold the largest number of anticipated objects. If that number is ten thousand, then forty kilo-bytes of space is needed as overhead to support the free-list (each handle is a four-byte integer).

A much simpler unique identifier facility was eventually adopted. Rather than use a free-list, a circular list was used with a pointer to the next available handle. Since the handle was implemented as an integer data type, the handle circular list was implemented as an integer list in the range **1..MAXINT**. The pointer starts at one end and works its way to the other. Since the maximum integer in Verdix Ada is 2_147_483_647, it is highly improbable that the pointer will ever circle back during the lifetime of the application. Furthermore, the previous issues of data

integrity and storage space are negligable. Since only the current pointer is needed, storage space is minimized and data integrity is reduced to managing a single record.

To implement this circular list, the following items were needed (their declarative names are shown in parenthesis):

- A variable to hold the circular list pointer (**next_handle**)
- A procedure to initialize the list pointer (**initialize_next_handle**)
- Procedures to load and save the pointer from and to secondary storage (**load_next_handle** and **save_next_handle**)
- Procedures to allocation and deallocate handles from the list (**allocate_handle** and **deallocate_handle**)

Only the second and third items are visable outside the object manager. The variable is maintained inside the manager. Allocation involves incrementing the pointer whenever an object is created. Deallocation currently does nothing. When an object repository is initially created, the person using the object manager calls the procedure to initialize the list pointer. The user is then responsible for saving the pointer as objects are created.

5.2.3 Object Repository An object repository or database was needed to organize, store, and manage the instantiated objects. The object repository is organized as an indexed-sequential list. Appendix M.2 contains the specification of the generic list package used by the object repository. This package contains all the necessary functions needed to initialize (**clear**), populate (**insert** and **delete**), access, **load**, and **save** an indexed-sequential list. Access to items in the list can be either sequential (**get_first** and **get_next**), reverse-sequential (**get_last** and **get_previous**), or random (**get** and **set**).

The list in this generic package is both a binary tree structure (the index) and a sequential list structure. When objects are added or removed from the list, the indexed-sequentiality is maintained. However, the binary tree structure may become unbalanced or lopsided. The re-

build_index procedure re-balances the tree in order to provide optimum random access to the list elements.

Each class in the data and drawing models has its own indexed-sequential list of instantiated objects. This guarantees that objects can only be accessed by class. To bring all these object lists together, a master object list was created. Each time an object is created or deleted, both the class' local list and the project's master list get updated. This master list is used by the dispatcher (described later) to route messages to and from individual classes. The master list contains the handle and class name for every object. The body of the object package in Appendix M.3 shows the master list. The function **get_class_name** takes an object handle, queries the master list, then returns the class name of the object.

5.9 Basic Class Implementation

The basic class was implemented as a generic package. Its code listing is found in the object package in Appendix M.3. Every class instantiates this basic class and inherits all its characteristics. Upon instantiation, each class supplies its class name, its attributes' record type, and the initial values of its attributes. For example, the activity class of Figure 25 would instantiate the basic class as shown in the following listing.

```
------------------------------------------------------------

   type activity_attributes_type is
      record
        name : string;
        code : string;
      end record;

   package activity_basics is new object_pkg.basics_pkg (
      class_name         => "activity_class",
      attributes_type    => activity_attributes_type,
      initial_attributes => (null_string, null_string));

------------------------------------------------------------
```

This basic class package provides all the operations shown in the system design model (see Figure 8). Comments in the code listings describe the implementation of these operations. Two procedures in the private body of the package require additional explanation. Upon first examining the procedures **initialize** and **finalize**, one might ask why they are needed. Why not just put them inside the **create** and **delete** operations? The reason for using these two procedures was derived from the Classic Ada User's Manual (26:98–100). These procedures are activated when objects are created and deleted. It is always good programming practice to initialize an object's values when it is created. If the object has a superclass, a message could be sent to the parent to have it initialize its values. Likewise, when an object is deleted, it is always good practice to "clean-up" before disappearing. For example, if an object has components, then they should also be deleted. Initialization and finalization are necessary to object instantiation.

5.4 Important Implementation Decisions

Before and during implementation, a few important decisions were made regarding the implementation of certain aspects of the system design. In some cases, these decisions were dictated by the limitations and restrictions of the Verdix Ada language environment. Other decisions were guided by performance, efficiency, "object-orientedness", and other related issues. This section discusses some of these implementation decisions.

5.4.1 Modeling Association as a Class Rumbaugh suggests a number of ways to implement associations (17:246–247). The method choosen for this research was to implement associations as classes. In this way, associations are independent from the classes they connect. If the associations were embedded as pointers in the respective classes, then when associations are added, changed, or deleted, the respective classes are affected. Implementing associations as classes removes this dependency. "This approach is useful for extending predefined classes from a library which cannot be modified, because the association object can be added without adding any attributes [pointers] to the original classes (17:247)."

Binary, ternary, and aggregation associations were all implemented as classes. Link attributes were also implemented as classes. Appendix M.5 contains code listings of these associative classes. A link is implemented as a pair (or triplet for ternary) of object handles. Roles were added to associations that required such attributes to distinquish the types of connected objects.

Access to an object's associated object is slightly less efficient for this method of implementing associations than using embedded pointers. This trade-off became an issue in the implementation of the drawing model where aggregates were derived from associations in the data model. The entire associative object list had to be searched to derive the correct links.

5.4.2 Encapsulating Class Attributes Ada provides the **private** declaration to restrict access to internal fields or structure of data types. As mentioned previously, private record types were originally planned to be used to encapsulate class attributes and restrict external access. However, dependencies in the Ada compiler prevented the use of private types. The following code listing illustrates this problem. The **attributes_types** is declared as a private record and fully declared later in the private section. The **object_basics_pkg** generic package is instantiated in order to inherit all the basic class attributes and operations. This code will not compile because Ada requires that types must be fully declared before using them as parameters in generic instantiations.

```
--------------------------------------------------------------------------------

with label_pkg;
with object_pkg;

package class is

   class_name : constant string := "class";

   type attributes_type is private;

   initial_attributes : constant attributes_type := (label_pkg.nul);

   package object_basics_pkg is new object_pkg.basics_pkg (
      class_name, attributes_type, initial_attributes);

private

   type attributes_type is record
      name : label_pkg.vstring;
   end record;

end class;

--------------------------------------------------------------------------------
```

The only solution to this problem was to duplicate the specifications for all the basic class operations in the package specification, instantiate the basic class in the package body, then redirect the duplicate operations to the basic class operations (see the following code listing). However, this solution violates the rules of inheritance and reuse by requiring the duplication of operations. For instance, if an operation is added to or removed from the basic superclass, then all subclasses would require changes. Therefore, because of this compiler dependency (private types need to be fully declared), all class attributes are public.

```
-------------------------------------------------------------------------------

with label_pkg;
with object_pkg;

package class is

   class_name : constant string := "class";

   type attributes_type is private;

   initial_attributes : constant attributes_type := (label_pkg.nul);

   procedure create (...);
   procedure delete (...);
   ...
   ...

private

   type attributes_type is record
      name : label_pkg.vstring;
   end record;

end class;

package body class is

   package object_basics_pkg is new object_pkg.basics_pkg (
      class_name, attributes_type, initial_attributes);

   procedure create (...) is
   begin -- create
      object_basics_pkg.create (...);
   end create;

   procedure delete (...) is
   begin -- delete
      object_basics_pkg.delete (...);
   end delete;

   ...
   ...

end class;

-------------------------------------------------------------------------------
```

5.5 Data Model Implementation

For each class shown in the system design diagrams of the data model, there is a corresponding class package (except for subclasses that were combined with their parent superclass). After

inheriting attributes and operations from the basic class, each class was extended as needed to include any additional attributes. Access operations (**get** and **set**) were also added for these new attributes. Finally, another attribute (and its associated access operations) was added for classes with subclass discriminators. An example of a class with subclasses is shown in Appendix M.4.

As mentioned earlier, all associations were implemented as classes. Associative class attributes include an array of object handles for the connected objects, and role attributes to distinquish the role of a connected object. For example, a state can perform an entry or exit action (see Figure 25). The object array holds the handles for the associated state and action objects, and the role attribute distinquishes whether the action is an entry or exit action. Appendix M.5 has code listings of sample associative class packages.

5.6 Drawing Model Implementation

The drawing model design is much more complex than the data model design due to all the derived aggregations and attributes. Initial implementation of the drawing model was attempted, but it was not completed due to time constraints.

5.7 Dispatcher Implementation

A dispatcher was built for this project to route messages to their respective classes. The dispatcher is basically a large **case** statement (actually implemented as an **if-then-elsif-...-else** statement). When a message arrives at the dispatcher, the message is first routed to the appropriate method or operation. Case statements in each method then decide which class gets the message. The following listing shows the code for a segment of the dispatcher.

```
procedure clear (
   object_name : in string) is

begin -- clear

   if object_name = data_pkg.argument.class_name then
      data_pkg.argument.clear;
   elsif object_name = data_pkg.association_parts.aggregation_name then
      data_pkg.association_parts.clear;
   elsif object_name = data_pkg.association.class_name then
      data_pkg.association.clear;
   elsif object_name = data_pkg.attribute.class_name then
      data_pkg.attribute.clear;
   elsif object_name = data_pkg.behavior.class_name then
      data_pkg.behavior.clear;
   ...
   ...
   ...
   else
      raise object_name_error;
   end if;

exception
   when others => raise;

end clear;
```

Because standard Ada does not provide function pointers, there is no way to implement dynamic message binding or dynamic method resolution. The dispatcher implemented for this thesis project is very "un-object-oriented"; when classes or class methods are created, deleted, or changed, the dispatcher must be modified. If Ada provided function pointers, methods and their respective classes would not need to be "hard-coded" in the dispatcher; they could be resolved at run-time. The dispatcher was modified more than any other code module in this thesis project.

5.8 Implementation Summary

During the implementation phase, some important issues emerged. This section addresses these issues. As will be seen, some issues required reassessments of the requirements analysis and system design phases. Although a "one-pass", water-fall software process was used during this

thesis, it became very apparent that a "multi-pass", spiral software process could have easily been used, if time had permitted.

5.8.1 Changes to the Basic Class Because all classes inherit the basic class and use the object manager, any changes to the basic class or object manager affected all other classes. Changes to the specification required a total recompilation. Changes to the body required a total update. In either case, it took the Verdix Ada compiler over one hour to regenerate the working driver program. Once the basic class and object manager were fully designed and implemented, this re-generation time reduced significantly.

It is understandable that a total recompilation should take a long time, but a total update should not take nearly as long. The advantages of separating package bodies and specs were not evident in the Verdix compiler. In fact, the total updates often aborted when temporary file space was exceeded at approximately ten mega-bytes. It was very apparent that this thesis project tested the limits of the Verdix compiler.

The size of this project will tremendously effect future efforts to enhance this project. A significant amount of code will be needed to support the drawing model and a graphical user interface such as MOTIF. Code re-generations could take hours under the current Verdix environment.

In a related issue, the size of the libraries needed to support the data model grew to over forty mega-bytes of disk space. Well over one-hundred mega-bytes of space will be needed to support future efforts to enhance this project.

5.8.2 Exception Handling Ada exception handling was used to catch error conditions. When using exceptions in an object-oriented environment, care must be taken to encapsulate and scope the raising and handling of exceptions. If exception handlers are not properly located, raised exceptions can violate the intended system performance. Exceptions are essentially additional parameters or

messages that are sent to the calling procedure when exceptions are raised. Therefore, exceptions must be well documented and not hidden in the code.

Exceptions should be planned and designed before implementation begins. The proper locations for exceptions in the OMT are in the dynamic and functional models. Exceptions could, and probably should, be designed as events. Raising an exception should cause one of the following:

- Send the exception event to another class to handle the exception
- Output the exception event to another state where it is handled
- Trap and handle the exception event inside the current state

Exception events should not cause transistions alone. Exception events must be tracable from their origin to a handler; otherwise, the exception will trigger an unexpected, unpredictable chain reaction, usually causing the application to crash.

Although Rumbaugh discourages excessive use of control flows in the functional model (17:129), exceptions are an exception! Because they are so important to track, exceptions should be modeled as control flows. When a process raises an exception, the control flow regulates and delegates control to the appropriate exception handling processes. Control flows in the functional model combined with events in the dynamic model are adequate ways to design exceptions.

VI. Conclusion and Recommendations

This chapter reviews the progress made during this research effort to answer the problems stated in Chaper I. Some interesting concepts and important conclusions were revealed. Hindsight also revealed some important lessons learned. Finally, with these new-found concepts and lessons learned, some recommendations are listed to help future efforts in this research area.

6.1 What was Accomplished?

This section addresses the level to which the goals of this thesis were achieved. In doing so, the questions in the problem statement are reviewed and answered. Next, the requirements are reviewed to consider how closely they were met. Finally, personal comments are made as to the success of this research effort.

6.1.1 Answering the Problem Statement All questions asked in the problem statement were addressed and answered at some point during this research. The following conclusions were made:

1. The OMT adequately represents basic object-oriented systems. All system designs were drawn using OMT object model notations. This author did not encounter any cases where the OMT object model was inadequate to build the design diagrams. The dynamic and functional models were not used during development; therefore, this author cannot properly judge whether these two submodels of the OMT are adequate. Some suggestions to extend the OMT model to allow more advanced modeling features are listed in a later section of this chapter.
2. The essential data elements of the OMT were described in detail in Chapters III and IV. A consolidated list of these elements is found in Appendix N. Each required data element on this list was designed and implemented as a class, association, or aggregation.

3. The data model developed by this research adequately describes, represents, and stores the essential data information of the OMT. Validation of this claim is shown by providing various instance diagrams in Appendix N.

4. The three sub-models of the OMT are cross-linked as described in Chapters III and IV. As stated in Chapter V, some of these cross-links are only conceptual, and therefore not implemented, while others are actual, implementable relations.

5. The graphical elements of the OMT are described in Chapters III and IV. A consolidated list of these elements is found in Appendix N.

6. The drawing information was abstracted into basic drawable elements and separated from the data information. No information in the drawing model is duplicated in the data model.

7. A "binding" mechanism was developed to combine the data and drawing models to create a logical and visual depiction of the OMT. Derived types in the drawing model derive the essential data information from the data model. These derived types bind the models together.

6.1.2 Meeting the Requirements The main requirement in this research was to design and implement a complete data and drawing model of the OMT. The major portion of this requirement was achieved. The data model was completely designed and implemented. However, the drawing model was completely designed but only partially implemented. Some of the reasons for this are:

- Many of the binding mechanisms were put into the drawing model. The exact locations and implementations for these mechanisms was sometimes difficult to "pin-down".
- Many of the binding mechanisms in the drawing model were derived attributes, associations, and classes which were derived from base elements in the data model. Each derived type required additional, sometimes complex, overhead to implement.

- The graphics-independent library requires routines implemented in the targeted environment. Originally, a MOTIF library was planned to be implemented, but the "learning curve" prohibited that from happening.

All other requirements were met during this research.

6.1.3 Concluding Comments The approach taken to solve this research problem contributed greatly to what was accomplished. The most complex part of the project was modeling the data elements of the OMT. Some of these difficulties are discussed in a later section. Because the data model was designed first, a thorough understanding of the OMT was gained early in the research. This helped tremendously during the requirements analysis and system design phases. If the drawing model had been implemented first, this understanding would have been much less.

During the design phase, it helped tremendously to take examples of real-world systems and see how well they could be modeled with the designs. Many examples were taken from Rumbaugh's text to show that they could be modeled with the data and drawing models.

6.2 Difficulties in Modeling Models

When first embarking on the task of modeling the OMT data and drawing elements, many of the difficulties were unknown. As in any scientific experiment, the process often reveals many other problems, questions, and side-issues. One important problem discovered in this research was the complexity and enormity of modeling the OMT model. This section discusses some of these issues.

Metamodels, or models of other models, "are less detailed, more fundamental, broader in scope, and often more difficult to understand (19:13)." Metamodels are very complex and occur infrequently, and they are a higher level of abstraction than the model itself. This concept is shown in Figure 9. The OMT models real-world objects. Such models are often very intuitive to the

analyst because the objects usually map directly to physical and conceptual entities. However, metamodel objects, also called metadata, are more likely to be abstract objects that are harder to comprehend because they may not map directly to objects in the real-world.

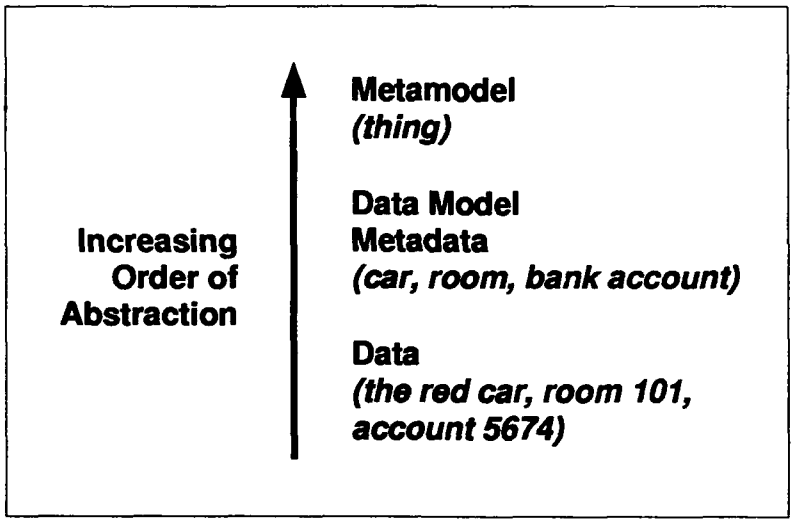


Figure 9. Abstraction Level

"Metadata is frequently confusing because it blurs the normal separations between the model and the real world (19:13)." Because the OMT itself (its notations and concepts) was used to describe and build the data and drawing models in this thesis, the development process often required very careful and intense conceptual thought. For example, it was easy to abstract the concept of an object class as an entity containing identity, state, and behavior. However, it was more difficult to analyze and abstract the characteristics of such concepts as behavior and inheritance, concepts that are already abstract by themselves.

Finally, Ada does not provide runtime access to variable names and function calls. Metadata attributes are often the variable and function names that the data is modeling. For example, the CLASS metadata type has the following structure:

```
type Class is record
    Name : LabelType;
    Attributes : AttributesType;
```

```
    Operations : OperationsType;
end record;
```

where the AttributesType is defined as:

```
type AttributesType is array (...) of record
    Name : LabelType;
    Type : LabelType;
    InitialValue : LabelType
end record;
```

If Ada allowed runtime access to variables (by means of a variable metadata type), class attributes could be implemented directly by the language instead of being abstracted into string label types. Blaha addresses this issue in his article and states, "making metadata available at runtime increases power and flexibility at the cost of complexity and possible loss of efficiency (19:14)."

6.3 Suggested Improvements to the OMT

Although Rumbaugh's OMT provides the basic elements and features needed to adequately describe an object-oriented system, some improvements can be made to the model itself to help it better represent these systems. Listed below are some of these recommended improvements.

- *Association Direction* - An association between two objects often implies a direction. Certain object models provide arrows to show the direction of an association, but in the OMT notation, this direction is only implied in the association's name. It is essential to understand this direction when implementing a model. The direction shows the way messages are routed between the connected classes, and helps show the implementer what *send* and *receive* methods are needed in these classes and associations. Therefore, the addition of a directed (or doubly directed) arrow on associations and links is recommended as an extension to the drawing

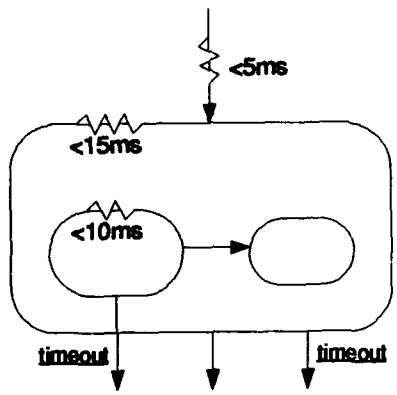


Figure 10. Timing Constraint Notation

model. The data model captures this directional information in the association classes (i.e., the *forward* and *inverse* roles in association connections).

- *Timing Features* - Timing characteristics are essential to a real-time system. The state-transition diagrams in the OMT dynamic model could be enhanced to include timing specifications such as those in the Harel Statechart methodology (7:800–801). Figure 10 shows an example of notations for timing constraints.

6.4 Recommendations for Implementation

Standard Ada is considered to be an object-based programming language (27:19+). This means it provides many object-oriented features such as encapsulation by using packages and reuse by using generics. However, because Ada lacks the capabilities to implement true inheritance and dynamic message binding, among other things, other languages and development tools should be considered when deciding what development environment and language to use to implement a system designed in an object-oriented fashion.

6.4.1 Other Programming Languages The object-oriented modeling tools developed for this thesis are perhaps slightly hampered because the underlying implementation platform (i.e., Ada) is not truly object-oriented. Some in industry feel that OOD tools should be "objects all the way

down", to eliminate the conceptual gaps that occur when an object-oriented tool is implemented with a conventional or non-object-oriented "back end" (21:95). For example, the data model of the OMT needed to support the generalization and inheritance concept. Since standard Ada lacks inheritance, creative thinking was required to develop code that could adequately abstract and represent inheritance. Therefore, the following recommendations could help reduce or eliminate the implementation problems discussed above.

- *Use Classic Ada* - Classic Ada is an extension to standard Ada that provides many missing object-oriented features (26). Most importantly, it provides:

  1. Class types or structures
  2. Inheritance of attributes and operations
  3. Dynamic binding (run-time method resolution)
  4. Message routing between classes and objects
  5. Object management

  Classic Ada was considered for use during this research effort, but was rejected because of the "learning curve", and it was not known if it could be used in the given development environment (i.e., Verdix Ada, SAMotif bindings). It is now known that Classic Ada is compatible with most Ada compilers and bindings, so early planning to learn Classic Ada would have helped tremendously in this research effort.

- *Use Ada 9X* - When released, Ada 9X will make it easier to implement object-oriented designs (1). For example, tagged record types can be used in inheritance to add new attributes to a subclass. Furthermore, function and procedure access types (pointers) can be used to dispatch a message or operation at run-time (dynamic binding). Although Ada 9X will not be a "true" object-oriented programming language, it will provide the necessary items needed to easily build class structures, implement inheritance, and allow dynamic binding. However, object management will not be provided by the language, so it will still need to be provided by the programmer or an object-oriented database (see next section).

- *Use another Object-Oriented Programming Language (OOPL)* - Languages such as C++ and Smalltalk are widely accepted and used in the object-oriented software development community. Numerous magazines and trade journals such as Dr. Dobbs, Object Magazine, and The Journal of Object-Oriented Programming are full of ads and reviews about C++ and Smalltalk. Many of these vendors provide high-level application development tools (CASE) with their products such as pre-built component or class libraries and GUI (graphical user interface) builders. Furthermore, many of these products allow the source code to be ported to a variety of platforms with very little or no change to the code. The requirements for this thesis did not allow this option.

6.4.2 Relational and Object Databases When the system design of the data model was implemented, this author soon realized that some of the operations were *access* operations (17:131). Access operations are trivial; they read and write (get, set, and list) the attributes of an object. They define the visibility of a class' attributes by performing authorization, range, and type checks. The only other types of operations in the data model were object instantiation routines (clear, create, and delete) and object persistance functions (load and save).

At the final count, the complete data model had over ninety different object classes and associations (modeled as classes). Access, instantiation, and persistance operations were written and debugged for each of these classes. This work and effort would have been significantly reduced by using either a relational (RDBMS) or an object database management system (ODBMS). RDBMSs and ODBMSs provide many more features than could possibly be implemented by this author. These facilities are (24:27-28):

1. **Data Model - class structures**
2. **Persistence - secondary storage and retrieval**
3. **Concurrency - multiple access to objects**
4. **Recovery - crash recovery**

5. Query Language - high-level access to attributes
6. Performance - efficient retrieval of objects
7. Security - access authorization
8. Data Abstraction - complex data and class structures
9. Object modeling - entities that capture both attributes and behavior
10. Identity - unique object instances
11. Encapsulation - data hiding
12. Inheritance - class reuse and extension through generalization and specialization
13. Polymorphism - operator overloading and dynamic message binding
14. Composition - aggregation (assembly) of objects from parts
15. Message passing - communication between objects to invoke behavior

"In short, the flexibility and power of object data representation combined with DBMS facilities provides a powerful medium for modeling, coordinated, storing, and manipulating ... information (24:28)."

6.5 Future Research

If it hadn't been properly defined and controlled, the scope of the thesis could have easily expanded beyond achievable results. There is enormous potential for future research efforts in this area. General Electric, the developers of the Object Modeling Techique, are working on solving many of these problems. Their OMTool product mentioned in this thesis is only a partial solution because it lacks the ability to model systems using the dynamic and functional models of the OMT. Therefore, the field is still "ripe" for research. A list of possible topics for future research is:

- Implement the data and drawing models of the OMT using an object-oriented environment such as Classic Ada or an object database system
- Build a graphical user interface front-end such as MOTIF or Open Windows
- Extend the data and drawing models to a specific domain model such as battlefield lines of communication or computer network models

- Map data and drawing models of other object methodologies such as Booch or Coad/Yoardon to the OMT
- Enhance the data and drawing models to allow real-time systems to be modeled
- Develop a smart tool to do consistency and completeness checking. This tool could also generate a visually optimized drawing model from an existing data model.

Appendix A. *OMT Notation Summary*

This appendix summarizes the notation used in Rumbaugh's Object Modeling Technique (OMT). These diagrams were taken directly from the inside front and back covers of Rumbaugh's text (17).

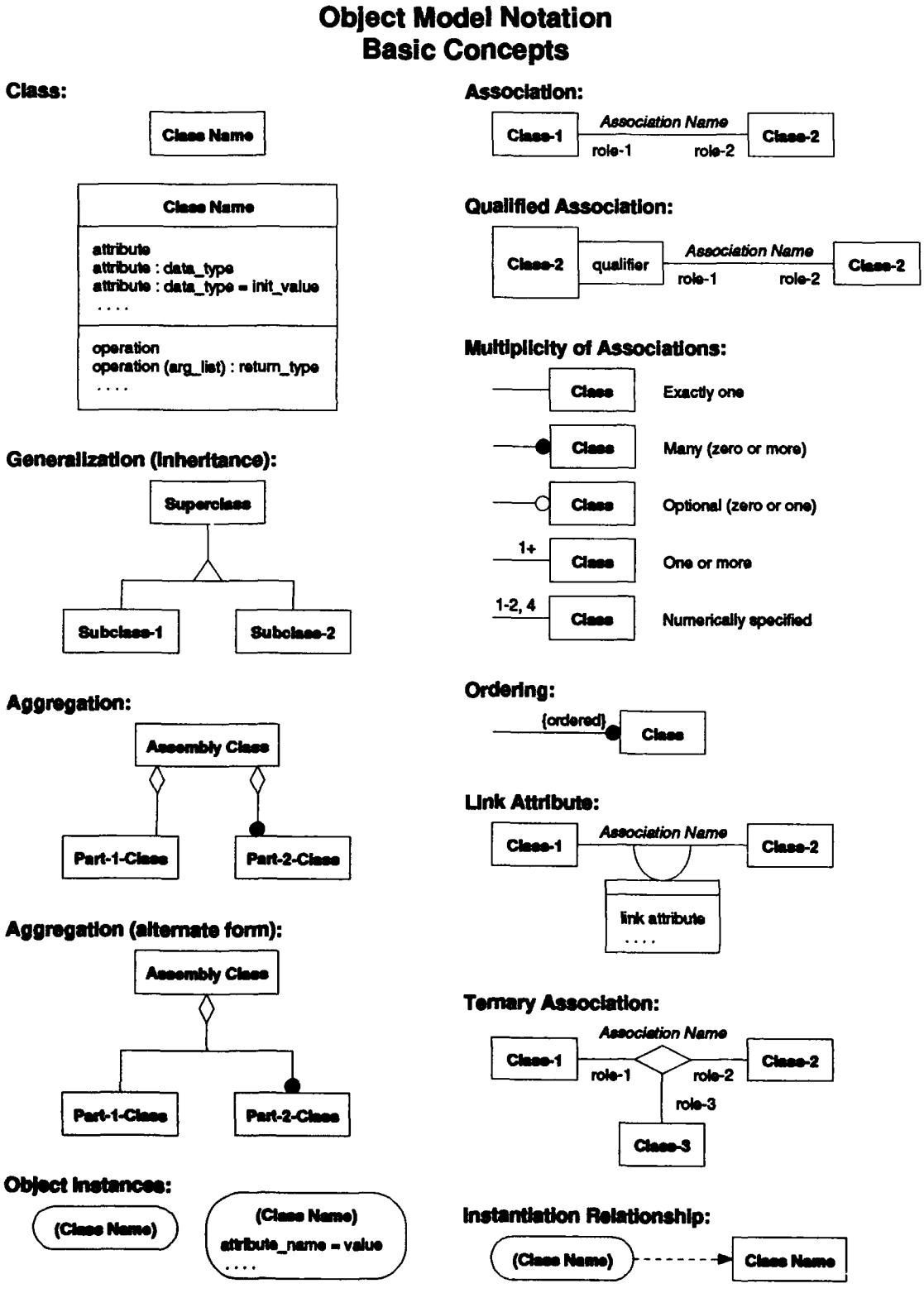


Figure 11. Object Model Notation: Basic Concepts

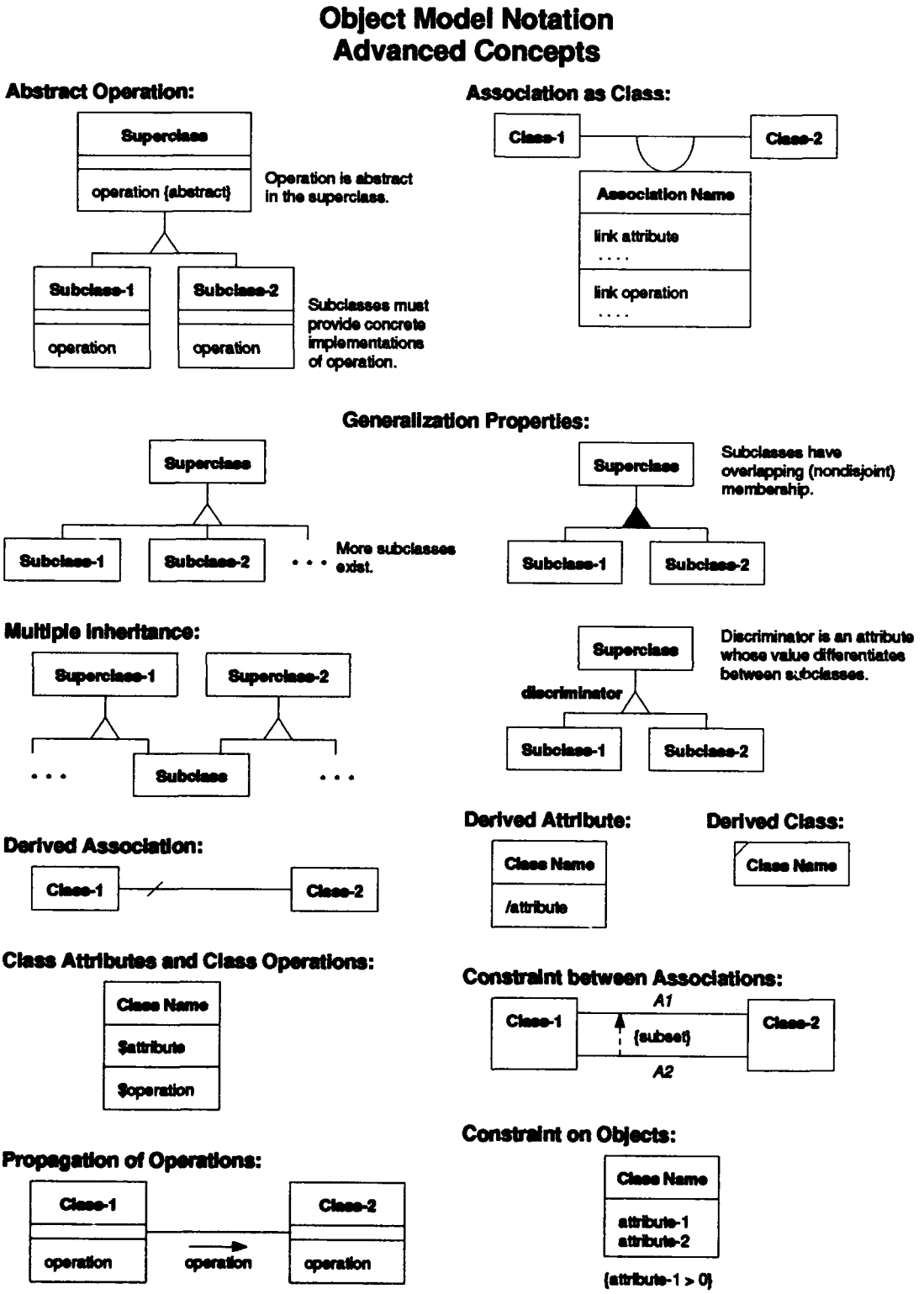


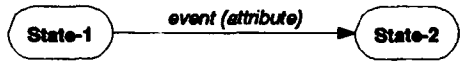
Figure 12. Object Model Notation: Advanced Concepts

Dynamic Model Notation

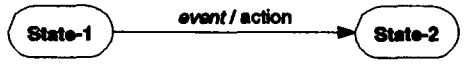
Initial and Final States:

Event causes Transition between States:

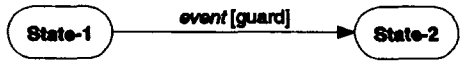
Event with Attribute:



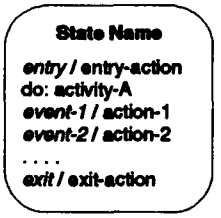
Action on a Transition:



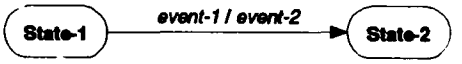
Guarded Transition:



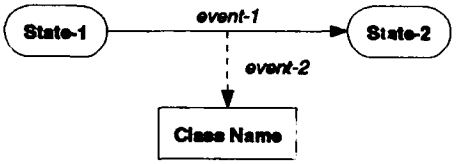
Actions and Activity while in a State:



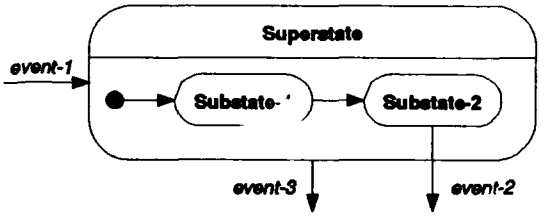
Output Event on a Transition:



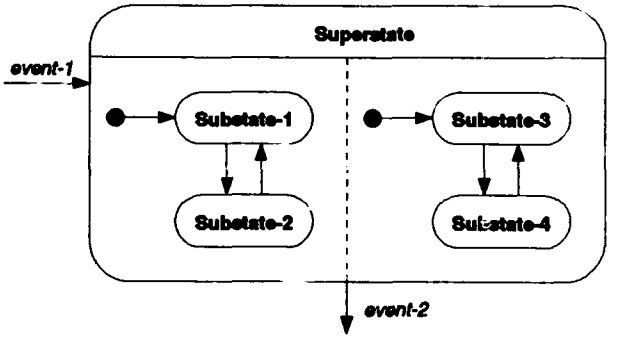
Sending an Event to another Object:



State Generalization (Nesting):



Concurrent Subdiagrams:



Splitting of Control:

Synchronization of Control:

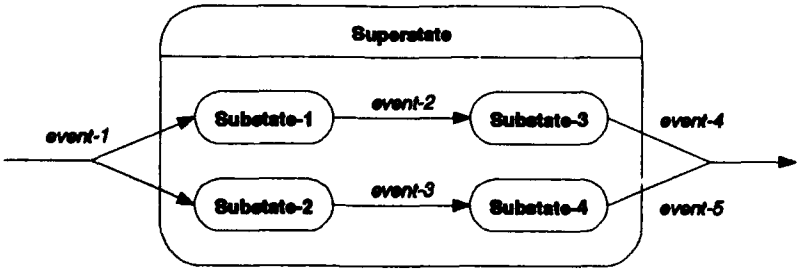


Figure 13. Dynamic Model Notation

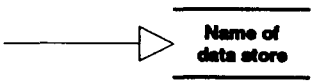
Functional Model Notation

Process:

Data Flow between Processes:

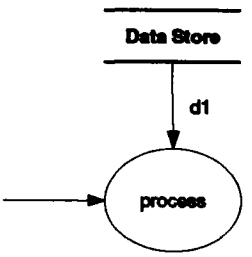
Data Store or File Object:

Data Flow that Results in a Data Store:

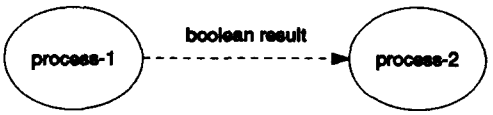


Actor Objects (as Source or Sink of Data):

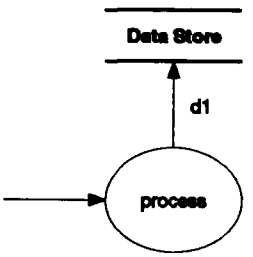
Access of Data Store Value:



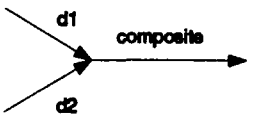
Control Flow:



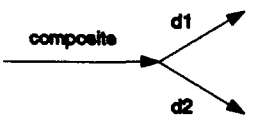
Update of Data Store Value:



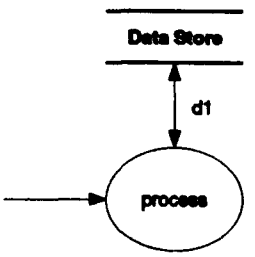
Composition of Data Value:



Decomposition of Data Value:



Access and Update of Data Store Value:



Duplication of Data Value:

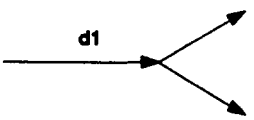


Figure 14. Functional Model Notation

Appendix B. *Detailed System Design*

B.1 Data Model Detailed Design

The data model design is shown in Appendix C. The data model class has a name. The NAME attribute is used to give a project name or title to the entire data model that is being created. The components of the data model are the three sub-models of the OMT. The three sub-models of the OMT are described with object, state, and data flow diagrams. The data element designs of each of these sub-models are discussed in the following subsections. In the last subsection of this section, the cross-links or relations between the three sub-models are explained.

B.1.1 Object Model Data Detailed Design The object model uses an entity-relationship methodology called object diagrams. Appendix D contains all the data model diagrams for the object model. The complete design of the object diagram is broken into nine modules. Each module is described below.

B.1.1.1 Class Design Module Figure 16 shows the *class* design module. A class is composed of zero or more attributes and operations. An attribute has a name and initial value, and an operation has a name. Operations are composed of an optional signature and behavior object. Behavior has a code block. A signature is composed of zero or more parameter arguments and an optional result argument. An argument has a name. Both attributes and arguments are defined for a specific domain, and domains can be defined for zero of more attributes and arguments. A domain consists of a type and a value set.

The BEHAVIOR object in the class structure provides an alternate way to describe behavior, rather than describing behavior in the dynamic and functional models. This encapsulates behavior in the object model and allows a complete object-oriented system to be designed using only the object model, behavior and all. This eliminates the sometimes complex and difficult task of relating and slicing the three submodels to capture class operations and behavior.

B.1.1.2 Inheritance Design Module Figure 17 shows the *inheritance* design module. Inheritance has two subclass: overlapping and disjoint inheritance. An attribute has an optional discriminator attribute. A discriminator attribute can be used to discriminate zero or more inheritances. Each inheritance is composed of connections to a superclass and a subclass. Connections are described later.

Two issues were considered when inheritance was designed. First, multiple inheritance is not shown explicitly in the design diagram. However, it can occur because an inheritance can be instantiated between two classes (superclass and subclass). Furthermore, overlapping inheritance can only occur in cases of multiple inheritance. Second, as shown in Figure 19 and described later, connections have multiplicity. In inheritance, this multiplicity is always one to one. This constraint was added to the inheritance design.

B.1.1.3 Association Design Module Figure 18 shows the *association* design module. This module consists of the association class and its related objects. An association has a name and is either an aggregation, binary, or ternary association. Aggregations have an assembler and component connection. Binary associations have a forward and inverse connection, and ternary associations have an additional middle connection. Associations can be modeled as classes or link attributes, and classes or link attributes can be models of zero or more associations. Link attributes are a subset of a class (class minus operations).

Orginally, aggregation was treated as a separate relationship from association. However, Rumbaugh states in his text that "aggregation is a special form of association (17:37)." Therefore, aggregation was changed to be a subclass of the association class.

B.1.1.4 Class Connection Design Module Figure 19 shows the *connection* design module. This module consists of the connection class and all its associated objects. A connection is composed of an optional role and required multiplicity. The role class has a name. A multiplicity

class has the subclasses one, many, optional, and specified. The many-multiplicity has an ordering attribute, and the specified-multiplicity has a description attribute. Connections connect to one class, and each class connects to zero or more connections. Multiplicity can be qualified by a qualifier attribute, and a qualifier attribute can qualify zero or more multiplicities.

It may be of interest to note that Blaha's object metamodel in Figure 5 represents connections differently. Although his design may be more compact, it lacks the ability to put an ordering on a many-multiplicity or to numerically specify a multiplicity. The connection design used for this thesis is a more complete representation of Rumbaugh's OMT.

B.1.1.5 Derives Design Module Figure 20 shows the *derives* design module. Three elements in the object model can have derived types. Derived classes can be derived from zero or more base or other derived classes, derived associations can be derived from zero or more base or other derived associations, and derived attributes can be derived from zero or more base or other derived attributes. The derived-from relations all have a derivation link-attribute.

In actual practice, derived types may derive from a variety of locations. For example, the CHILD class is derived from the MARRIAGE association between male and female PEOPLE classes. The derivation attribute is used to show when derivations occur from different types (i.e., classes from associations, attributes from classes). A derived-from association with mixed-type derivations is not shown in the design diagrams because it was impossible to draw. However, the implementation of the derives-from association will allow mixed-type derivations.

B.1.1.6 Constrains Association, Propagates Operation, and Operation Types Design Module Figure 21 shows the it constrains association, it it propagates operation, and it operation types design modules. An operation can be one of three types: normal, class, or abstract operation. An operation can be propagated by zero or more associations, and an association can propagate zero

or more operations. One association can be constrained by another association. The association constraint has a constraint link-attribute.

An association can actually only propagate a normal operation, but as can be seen in the design figure, the PROPAGATES association is connected to the OPERATION superclass. Because Ada does not support inheritance, superclasses are concrete but subclasses are not (subclasses are implemented as TYPEs). Therefore, although the design would be logically more correct if the PROPAGATES relationship was drawn between the normal-operation and association classes, the current design more correctly depicts the actual implementation.

B.1.1.7 Constrains Class, Attribute Types, and Class Types Design Module Figure 22 shows the it constrains class, it attribute types, and it class types design modules. An attribute has two subclasses: normal and class attributes. A class has the subclasses concrete and abstract. Zero or more attributes in a class can be constrained, and a class can have zero or more attribute constraint sets. The CONSTRAINS association has a constraint link-attribute.

The class constraint could have actually been designed on the aggregate relationship between CLASS and ATTRIBUTE in Figure 16. Instead, a separate class constraint association was created to single out this important concept. As is always the case in design, numerous correct interpretations are possible.

B.1.1.8 Instantiates Design Module Figure 23 shows the it instantiates design module. It shows all the object model elements that can be instantiated. A class instantiates zero or more objects, an attribute instantiates zero or more values, and an association instantiates zero or more links. The instantiated objects, values, and links all have ID attributes. In addition, values have a VALUE attribute. An object is composed of zero or more values, and connects to an optional connection. Links are composed of zero-to-two connections.

Originally, instantiation was not considered to be a data element of the object model. However, after further consideration, it was decided that instantiation must be included. Instantiation is a powerful tool to verify design correctness. For example, a designer creates object diagrams for his system. To verify that the design works, the designer takes a real-world example and instantiates it using the developed diagrams. This is analogous to a programmer working through his program with actual data. The need for instantiation became very apparent when real-world examples were used to validate all the designs for this thesis.

B.1.1.9 Object Diagram Design Module Figure 24 shows the it object diagram design module. Each object diagram has a name and is composed of zero or more class and instance diagrams. Class and instance diagrams both have names and are both composed of zero or more modules. Modules have names and are composed of zero or more sheets. Depending on the diagram (class or instance), sheets are composed of zero or more classes, associations, and inheritances, or zero or more objects and links.

It should be obvious from looking at this design that it is impossible to tell which module and sheets are part of the class diagram and which are part of the instance diagram. In actual practice, the two diagrams can be a mixture of class and instance elements. For example, Figure 11 shows the instantiation relationship. This is a case where it is necessary to allow classes and objects to exist on the same sheet. Therefore, sheets are actually composed of zero or more of any of the five components shown in the design figure.

B.1.2 Dynamic Model Data Detailed Design The dynamical model uses a state-transition methodology. Appendix E contains all the data model diagrams for the dynamic model. The complete design of the state-transition diagram is broken into four modules: the state, transition, connection, and state-diagram designs. Each module is described below.

B.1.2.1 State Design Module Figure 25 shows the it state design module. It is composed of the state object and its associated control objects. The state object has a name and three subclasses: initial, intermediate, and final states. The control objects, activity, action, and event, can only be associated to the intermediate state. Both an activity and an action have a name and code block. An event has a name. A state controls an optional activity, and an activity can be controlled by zero or more states. A state performs zero or more entry and exit actions, and actions can be performed by zero or more states. A state can trap zero or more events and perform zero or more internal actions. Finally, an activity and an action can generate zero or more events, and an event can be generated by zero or more activities and actions.

The TRAPS-PERFORMS association was modeled as a ternary relationship. All possible binary combinations were explored, but all failed to provide a good design of an event trap inside a state. For example, an event-trap binary association from the state to the event object, and from the event to the internal action object could not uniquely identify which events performed which internal actions for a specific state.

B.1.2.2 Transition Design Module Figure 26 shows the it transition design module. It is composed of the transition object and its associated attached objects. A transition has no attributes. The objects that can attach to a transition are external actions, guard conditions, and events. Actions and events are the same as described above. Conditions have a Boolean expression. An event can convey zero or more event attributes, and an event attribute can be conveyed by zero or more events. A condition guards zero or more transitions, an a transition can be guarded by zero or more conditions. A transition performs zero or more external actions, and an external action can be performed by zero or more transitions. An event can cause zero or more different transitions, and a transition can be caused by zero or more different events. A transition can output zero or more events, and an event can be output during zero or more transitions. A transition can send zero or more events to zero or more classes.

The SENDS-TO relationship was modeled as a ternary association for the same reason that the TRAPS-PERFORMS association was modeled as a ternary association. All combinations of binary associations failed to adequately represent a good design of this association. Also, an outputed event on a transition is an alternate way for actions to generate events. The only difference is that outputed events show the state destination for the generated events.

B.1.2.3 Transition Connection Module Figure 19 shows the it transition connection design module. It consists of all the associations needed to connect a transition. Transitions can connect from any state, superstate, split, and synchronization to any of the same. A state or superstate can have zero or more transitions going from or coming to them. A split has zero or one transitions going to it and zero or more transitions coming from it. A synchronization has zero or more transitions going to it and zero or one transition coming from it. Splits and synchronizations don't have any attributes, and superstates are described in the next section.

Connections are the "glue" that bind together the various elements of the state diagram. In a complete and consistent state diagram, each element must be connected to one or more transitions. Furthermore, splits must have two or more transitions coming out, and synchronizations must have two or more transitions going into them. However, during the building process, sometimes all states are designed before they are connected to transitions. In such cases, inconsistencies are unresolved until the latter stages of design. The connection design allows this condition to occur. That is why a "smart tool" is needed to check consistencies after the designs are complete.

B.1.2.4 State Diagram Module Figure 28 shows the it state diagram design module. It shows the elements that compose state diagrams, and it models state and event generalization. Each state diagram has a name and is assembled from zero or more transitions, states, superstates, splits, synchronizations, and state subdiagrams. State subdiagrams allow state diagrams to be nested. A superstate has a name attribute and is specialized into two subclasses: concurrent and generalized superstates. A concurrent superstate consists of zero or more state subdiagrams (two

or more in a complete and consistent design), and a generalized superstate consists of zero or one state subdiagrams (one in a complete and consist design). Event generalization occurs by allowing a sub-event to inherit zero or more super-events, and a super-event to be inherited by zero or more sub-events. State generalization occurs through using nested state diagrams or by using generalized superstates. Finally, events and activities can be expanded to state diagrams, and state diagrams can be expansions of zero or more events and activities.

B.1.3 Functional Model Data Detailed Design The functional model uses a data flow methodology. Appendix F contains all the data model diagrams for the functional model. The complete design of the data flow diagram is broken into three modules: two flow connection designs and a data flow diagram design. Each module is described below.

B.1.3.1 Flow Connection Modules Figure 29 and Figure 30 show the it flow connection designs. These designs consist of the various data flow elements and the associated connections between them. A data flow connects from and to any of the various objects shown in the figures. A data flow can connect from a source actor and/or to a sink actor, from an output data store and/or to an input data store, and from a process and/or to a process. Furthermore, as shown in the second figure, a data flow can connect from a duplication, composition, or decomposition, to any of the same. Store flows result from a process and result in a data store. A control flow regulates control from one process to another. Each control flow has zero or more guard conditions, and a guard condition can be attached to zero or more control flows.

When modeling the data flow diagram, a decision was made to model duplications, compositions, and decompositions in the data model. It was originally thought that these elements only belonged in the drawing model because they were merely convenient drawing constructs. However, it was determined that duplicating, composing, and decomposing data flows may actually require some form of processing at the implementation level, and therefore, they were included in the data model.

B.1.3.2 Data Flow Diagram Module Figure 31 shows the it data flow diagram (DFD) design. A DFD is composed of zero or more of any of the following: actors, processes, data stores, data flows, store flows, control flows, duplications, compositions, and decompositions. Actors, processes, data stores, and data flows all have names. Additionally, processes have code blocks, and data stores have file names, both of which are needed to model implementation issues. A process has two subclass: atomic and complex processes. A complex process can expand to a DFD sublevel, and a DFD can be an expansion of zero or more complex processes.

B.2 Drawing Model Detailed Design

Figure 37 in Appendix I shows the top-level drawing components of the OMT. The drawing model group is composed of optional data model text and zero or more object, state, and data flow diagram groups. The data model text has a name that is derived from the data model class in the data model. The following subsections describe each of the three diagram groups in the drawing model group.

B.2.1 Object Model Drawing Detailed Design Appendix J contains all the drawing model diagrams for the object model. The object model drawing design consists of nine modules: class group, inheritance group, association group, connection group, derivation group, constraint group, propagates group, instantiates group, and object diagram group. The following subsections describe each module in detail.

B.2.1.1 Class Group Module Figure 38 shows the drawing elements of the class group. The class group is composed of a class rectangle, class text (bold), optional attribute and operation list groups, and an optional class constraint group. The class text derives its name attribute from CLASS.NAME in the data model. The class constraint aggregate is derived from the constrains association in the data model. Constraint groups are described in a later subsection.

The attribute list group was created as a convenient way to co-locate all the attributes of a particular class into one list or region. Rumbaugh suggests this grouping technique in his text (17:237-238). This list group is composed of zero or more single attributes called attribute groups. An attribute group can be either a normal, class, or derived attribute. The group is composed of attribute text and optional domain text. Attribute text has two attributes (name and initial value) which are derived from the attribute class in the data model. Domain text has two attributes (type kind and value set) which are derived from the domain class in the data model. The domain text aggregate is derived from the defines association in the data model.

Like the attribute list, the operation list group was also created as a way to group singular operations. The operation list group is composed of zero or more operation groups. An operation group can be either a normal, class, or abstract operation. Operation groups are composed of operation text, zero or more parameter agrument groups, and an optional result argument group. Operation text has name and code attributes which are derived from the operation and behavior classes in the data model. The argument group is composed of domain text and argument text. The domain text is the same as described above. Argument text has a name attribute which is derived from the argument class in the data model.

B.2.1.2 Inheritance Group Module Figure 39 shows the drawing elements of the inheritance group. The inheritance group is composed of an inheritance node group, an inheritance arc, optional discriminator attribute text (bold), and both subclass and superclass connection groups. The discriminator has a name attribute which is derived from the discriminator attribute class in the data model. The connection group is described in a later subsection.

The inheritance node group is composed of an optional inheritance yoke and an inheritance polygon. The inheritance polygon has two subclasses that are derived from the inheritance subclasses in the data model. These subclasses are overlapping and disjoint inheritance triangle-shaped

polygons. The overlapping inheritance triangle is filled, and the disjoint inheritance triangle is empty.

B.2.1.3 Association Group Module Figure 40 shows the drawing elements of the association group. The association group is composed of an association arc group, optional association text, an optional propagates group, and an optional link attribute or association class group. The association text is italics style and has a name attribute which is derived from the association class in the data model. The propagates group aggregate is derived from the propagates assocation in the data model. This group is described in a later subsection.

The link attribute or association class group aggregate is derived from the MODELED-AS association in the data model. Link attributes are a subset of a class (i.e., class minus operations). In addition to all the aggregates of the standard class group, associations modeled as classes or link attributes also have a MODELED-AS half-circle node.

An association arc group is either an aggregation, binary, or ternary assocation group. These subclasses are derived from the association class in the data model. The aggregation association group is composed of an optional aggregation yoke and two connection groups (assembler and component). The assembler connection tip is a diamond-shaped polygon. The binary association group is composed of a forward and an inverse connection group. The ternary assocation group is composed of a ternary association diamond-shaped polygon and three connection groups (forward, middle, and inverse). The connection group is described in the next subsection.

B.2.1.4 Connection Group Module Figure 41 shows the drawing elements of the connection group. The connection group is composed of optional role text, a multiplicity group, and a class group. The class group aggregate is derived from the connects association in the data model. Role text has a name attribute that is derived from the role class in the data model.

The multiplicity group has four subclasses that are derived from the multiplicity class in the data model. These subclasses are the one, many, optional, and specified multiplicity groups. Currently, the one-multiplicity group has no graphical elements. The many-multiplicity group is composed of a filled circle node and ordering text. The ordering attribute is derived from the many-multiplicity class in the data model. The optional-multiplicity group is composed of an empty circle node. The specified-multiplicity group is composed of description text which is derived from the specified-multiplicity class in the data model. Finally, a multiplicity group is also composed of optional qualifier attribute text. This aggregate is derived from the qualifies association in the data model, and the name attribute in the qualifier text is also derived from the data model.

B.2.1.5 Derivation Group Module Figure 42 shows the drawing elements of the derivation group. This group is composed of a derivation arc (slash line) and optional derivation text. The derivation text attribute (derivation) is derived from the DERIVES-FROM.DERIVATION link attribute in the data model. The derivation group aggregation is derived from the DERIVES-FROM association in the data model.

B.2.1.6 Constraint Group Module Figure 43 shows the drawing elements of the constraint group. This group is either a class or association constraint group. The class constraint group is composed of constraint text. The association constraint group is composed of a dashed-line constraint arc and constraint text. The constraint attribute in the constraint text is derived from the CONSTRAINS.CONSTRAINT link attribute in the data model.

B.2.1.7 Propagates Group Module Figure 44 shows the drawing elements of the propagates group. The propagates group is composed of a propagates arc and operation text. The operation text has a name attribute which is derived from the operation class in the data model.

B.2.1.8 Instantiates Group Module Figure 45 shows the drawing elements of the instantiates group. This group is composed of three connected sub-groups: the link, object, and

value groups. The link group is composed of optional association text, a link arc, and two or three connection groups (two for aggregation and binary association, and three for ternary association). The association text is italics style and has a name attribute which is derived from the association class in the data model. This optional aggregate is derived from the instantiates association in the data model. The connection group is composed of an object group which is also derived from the instantiates association.

The object group is composed of a rounded rectangle, class text, an optional value list group, and an optional instantiation relation group. The class text is boldface and is derived from the instantiates association in the data model. Its name attribute is derived from the CLASS.NAME in the data model. The instantiation relation group is composed of a dashed-line arc and a class group. This class group aggregate is derived from the data model.

The value list group is composed of zero or more value groups. The value group is composed of attribute text and value text. The attribute text component is derived from the instantiates assocation in the data model. The attributes of the attribute and value text (name and value) are derived from the corresponding classes in the data model.

B.2.1.9 Object Diagram Group Module Figure 46 shows the drawing elements of the object diagram group. This group is composed of optional object diagram text, and zero or more class and instance diagram groups. The object diagram text has a name attribute which is derived from the object diagram class in the data model.

The class diagram group is composed of zero or more module groups, which are composed of zero or more sheet groups, and which are composed are zero or more class, assocation, and inheritance groups. The instance group is similarly composed except its sheet groups are composed of zero or more object and link groups.

A module group is also composed of optional module text, and a sheet group is also composed of optional sheet text. The attributes of these two text classes (name and page) are derived from module and sheet classes in the data model.

B.2.2 Dynamic Model Drawing Detailed Design Appendix K contains all the drawing model diagrams for the dynamic model. The dynamic model drawing design consists of four modules: state group, transition group, superstate group, and state diagram group. The following subsections describe each module in detail.

B.2.2.1 State Group Module Figure 47 shows the drawing elements of the state group. A state group is composed of a state node, optional state text, and an optional control group. State nodes are specialized into three subclasses: initial state circle (solid fill), intermediate state rounded rectangle, and final state group (outer empty circle and inner filled circle). The subclass is derived from STATE.SUBCLASS in the data model. State text has a name that is derived from STATE.NAME in the data model. The style attribute for state text is constrained to boldface.

The control group is composed of optional activity text, zero or more entry and exit action text, and zero or more event trap groups. The control group aggregation is derived from the corresponding associations in the data model. The activity text component is derived from the CONTROLS association, the action text components are derived from the PERFORMS association, and the event trap group is derived from the TRAPS-PERFORMS association. Both the activity and action text have name and code attributes. These attributes are derived from the corresponding ACTIVITY and ACTION class attributes in the data model. An event trap group is composed of an event group and zero or more internal actions. Event groups are further describing in the next subsection.

B.2.2.2 Transition Group Module Figure 48 shows the drawing elements of the transition group. A transition group is composed of a transition arc (solid line), an optional sent event group, and an optional event transition group.

A sent event group is composed of a sent event arc (dashed line) and zero or more classes and event groups. The sent event group aggregation is derived from the SENDS-TO associations in the data model.

An event transition group is composed of zero or more event groups, condition text, and external action text. The event transition group aggregation is derived from the corresponding associations in the data model. The event group components are derived from the CAUSES and OUTPUTS associations, the condition text components are derived from the GUARDS association, and the external action text components are derived from the PERFORMS association. The condition text expression attribute is derived from CONDITION.EXPRESSION in the data model.

An event group is composed of event text and zero or more event attribute texts. The event group aggregation is derived from the CONVEYS association in the data model. Event text has a name that is derived from EVENT.NAME in the data model. The attributes of the event attribute text, name and initial value, are derived for the attributes in the ATTRIBUTE class in the data model.

B.2.2.3 Superstate Group Module Figure 49 shows the drawing elements of the superstate group. A superstate group has two subclasses: generalized and concurrent superstate groups. These subclasses are derived from the corresponding discriminator in the data model.

A generalized superstate group is composed of a generalized superstate rounded rectangle, optional superstate text, and an optional state subdiagram group. A concurrent superstate group is composed of a concurrent superstate rounded rectangle, optional current superstate text, and zero or more state subdiagram groups. Both generalized and concurrent superstates have names (bold) that are derived from the corresponding classes in the data model.

B.2.2.4 State Diagram Group Module Figure 50 shows the drawing elements of the state diagram group. A state diagram group is composed of an optional state diagram rectangle, optional state diagram text (bold), and a transition connection group. The state diagram text has a name that is derived from STATE-DIAGRAM.NAME in the data model.

A transition connection group is composed of zero or more transition connection tuples, transition groups, state groups, superstate groups, split points, and synchronization points. The transition connection group aggregation is derived from the CONNECTS association in the data model. The transition connection tuple has two attributes: from and to. These attributes are derived from the roles on the CONNECTS associations in the data models. They are used to pair-up the components in this aggregation.

B.2.3 Functional Model Drawing Detailed Design Appendix L contains all the drawing model diagrams for the functional model. The functional model drawing design consists of four modules: node groups (process, actor, and data store groups), arc groups (data and control flow groups), flow connection groups, and data flow diagram groups. The following subsections describe each module in detail.

B.2.3.1 Node Groups Module Figure 51 shows the drawing elements of the functional model node groups. The three groups in this modules are:

- *Process Group* - The process group is composed of a process elipse node and optional process text (normal style). Process text has name and code block attributes which are derived from the corresponding PROCESS attributes in the data model.
- *Data Store Group* - The data store group is composed of a data store node group and optional data store text (bold). Data store text has name and file name attributes which are derived from the corresponding DATA STORE attributes in the data model. The data store node group is composed of an upper and lower yoke or line.

- *Actor Group* - The actor group is composed of an actor rectangle and optional actor text (bold). Actor text has a name attribute which is derived from the corresponding ACTOR.NAME in the data model.

B.2.3.2 Arc Groups Module Figure 52 shows the drawing elements of the functional model arc groups. The two groups in this module are:

- *Data Flow Group* - The data flow group is composed of a data flow arc (solid line) and optional data flow text (normal style). Data flow text has a name attribute with is derived from DATA-FLOW.NAME in the data model.
- *Control Flow Group* - The control flow group is composed of a control flow arc (dashed line) and zero or more condition text (normal style). Condition text has an expression attribute which is derived from CONDITION.EXPRESSION in the data model. The condition text components of the control flow group aggregation are derived from the GUARDS association in the data model.

B.2.3.3 Flow Connection Groups Module Figure 53 shows the drawing elements of the flow connection groups. The purpose of the connection tuples is to pair-up the components in the respective group. The three groups in this module are:

- *Data Flow Connection Group* - The data flow connection group is composed of zero or more data flow connection tuples, actor groups, data flow groups, data store groups, process groups, and flow junction points (duplication, composition, and decomposition). The data flow connection group aggregation is derived from the CONNECTS association in the functional data model. The data flow connection tuple has a from and to attribute which are derived from the roles on the CONNECTS association.
- *Store Flow Connection Group* - The store flow connection group is composed of zero or more store flow connection tuples, data store groups, process groups, and store flow arcs (solid line).

The store flow connection group aggregation is derived from the RESULTS-IN association in the functional data model. The store flow connection tuple has a from and to attribute which are derived from the roles on the RESULTS-IN association.

- *Control Flow Connection Group* - The control flow connection group is composed of zero or more control flow connection tuples, process groups, and control flow groups. The control flow connection group aggregation is derived from the REGULATES association in the functional data model. The control flow connection tuple has a from and to attribute which are derived from the roles on the REGULATES association.

B.2.3.4 Data Flow Diagram Group Module Figure 54 shows the drawing elements of the data flow diagram (DFD) group. A data flow diagram group is composed of optional DFD text, a complex process group, and data flow, store flow, and control flow connection groups. DFD text has a name which is derived from DFD.NAME in the data model. A complex process group is composed of a DFD sublevel group. This last aggregation is derived from the EXPANDS-TO association in the functional data model.

Appendix C. *Data Model Design Diagram*

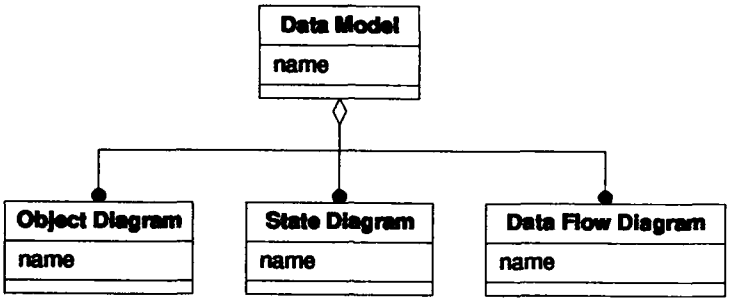


Figure 15. Data Model Design

Appendix D. *Data Model: Object Model Design Diagrams*

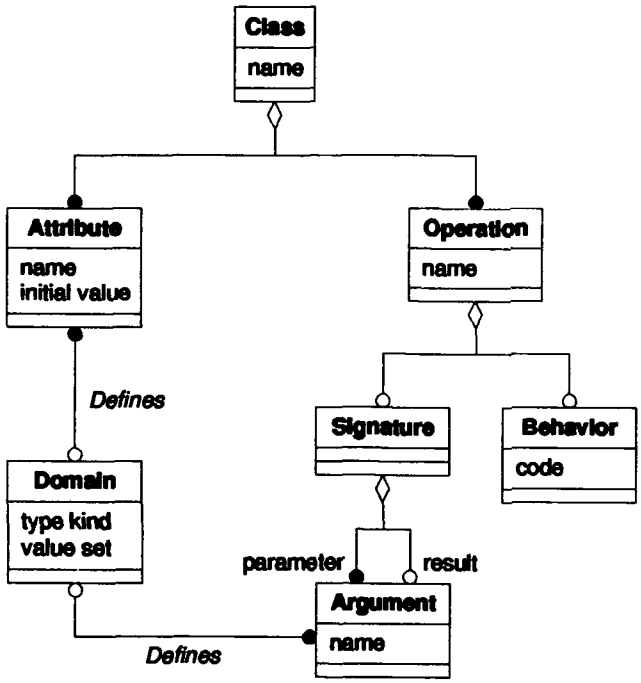


Figure 16. Class Design

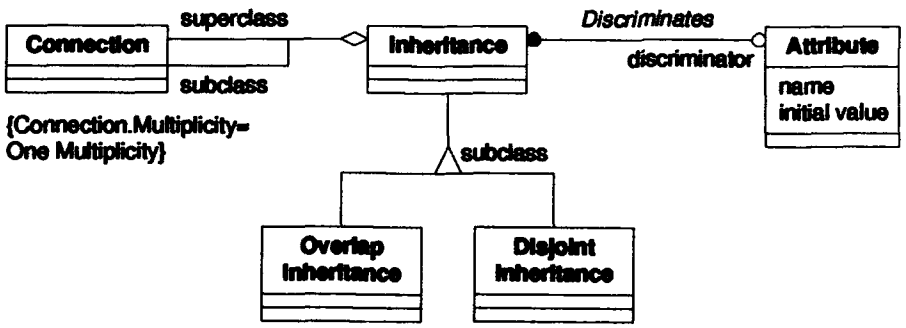


Figure 17. Inheritance Design

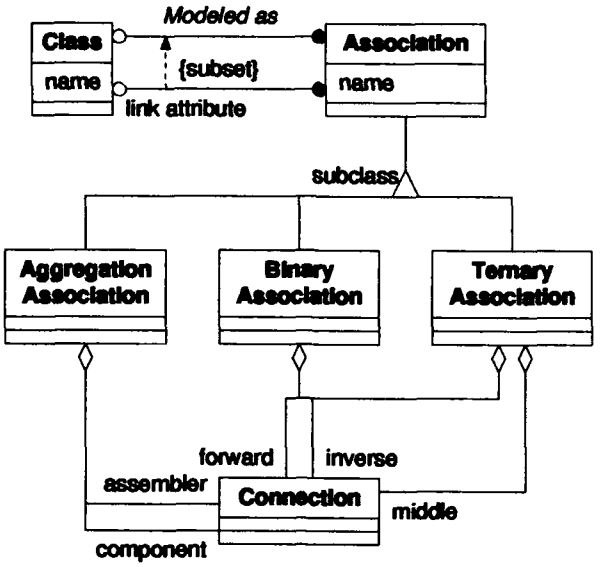


Figure 18. Association Design

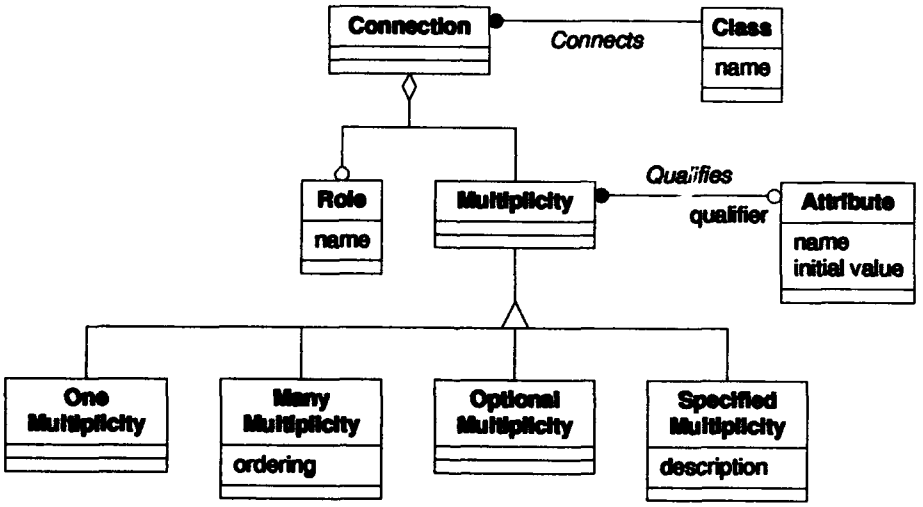


Figure 19. Connection Design

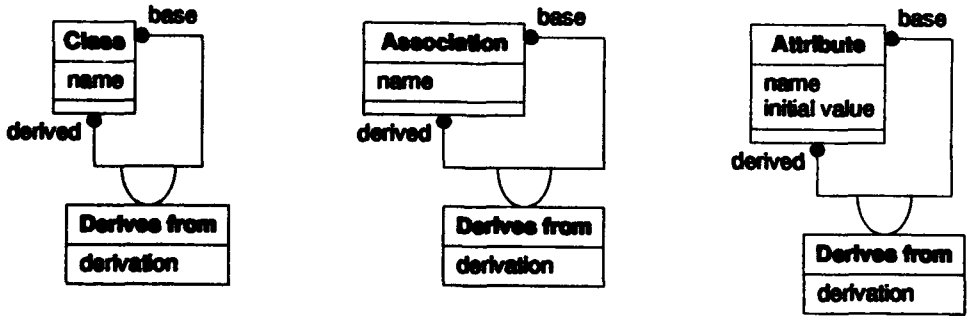


Figure 20. Derives Design

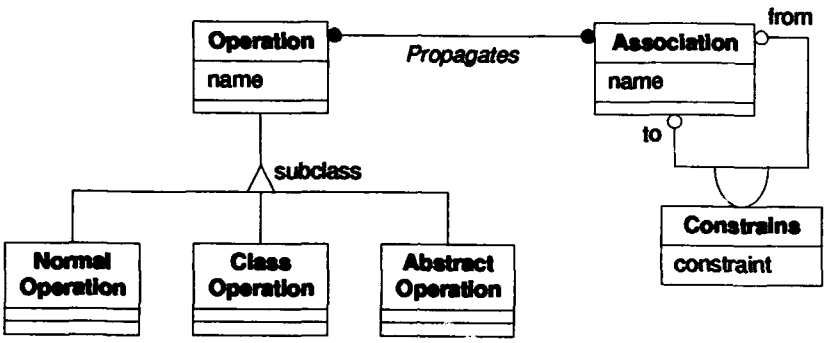


Figure 21. Constrains Association, Propagates, and Operation Types Design

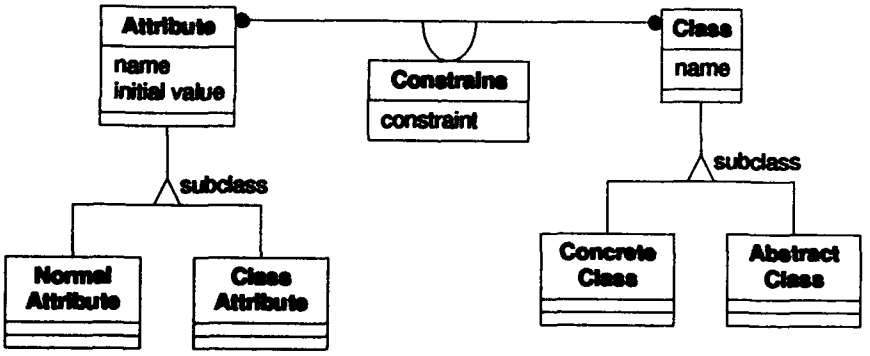


Figure 22. Constrains Class, Attribute Types, and Class Types Design

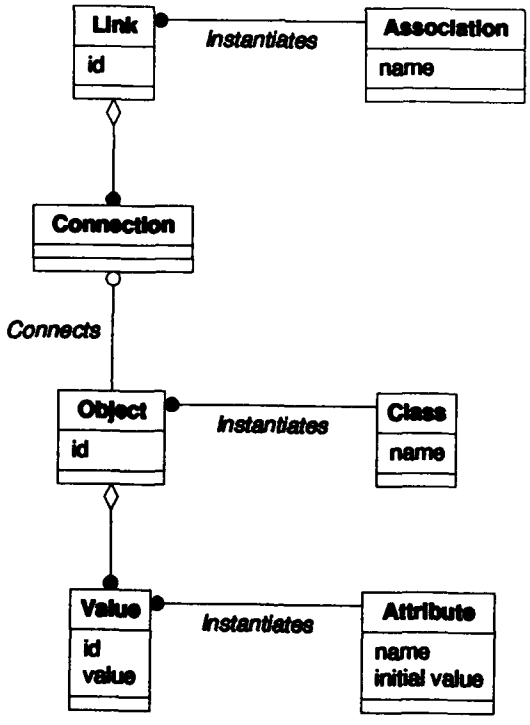


Figure 23. Instantiates Design

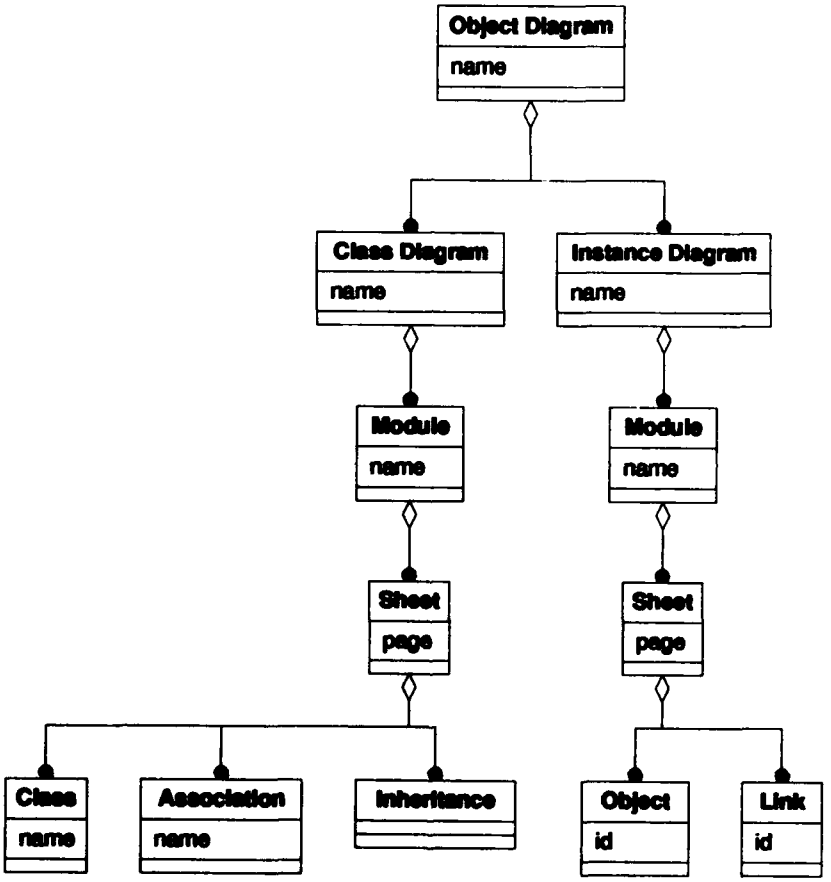


Figure 24. Object Diagram Design

Appendix E. *Data Model: Dynamic Model Design Diagrams*

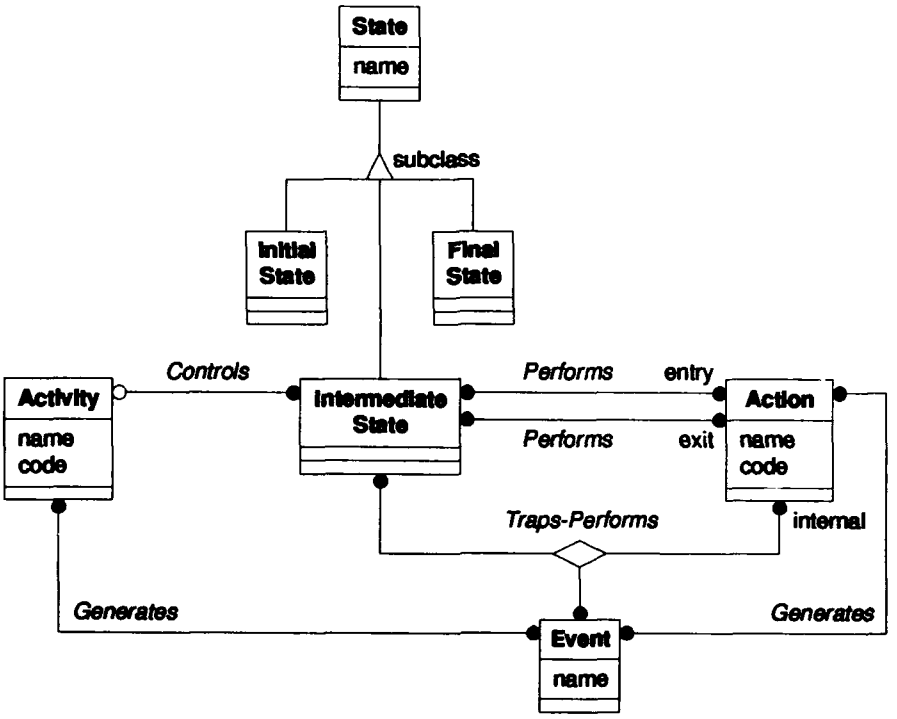


Figure 25. State Design

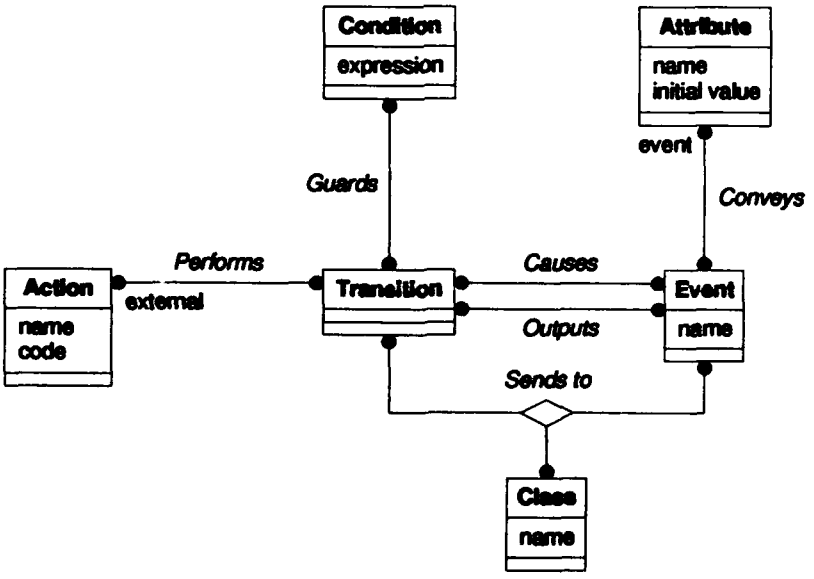


Figure 26. Transition Design

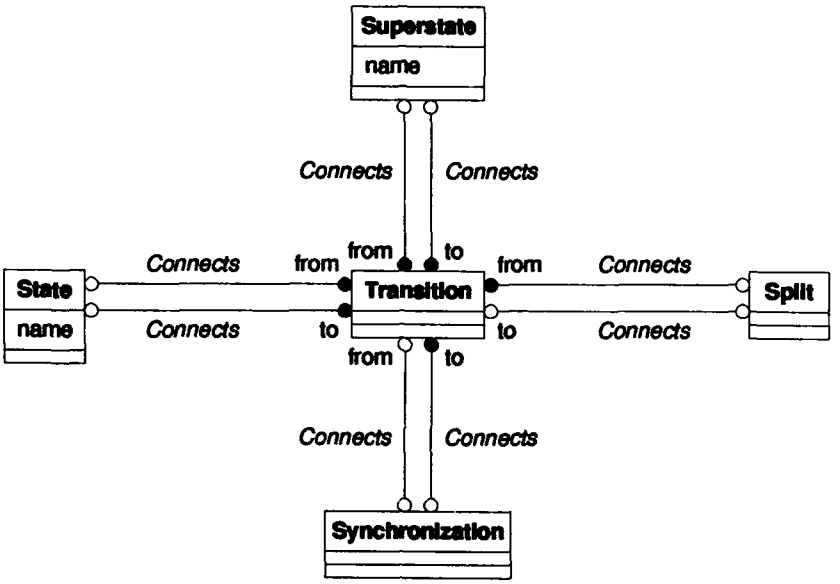


Figure 27. State-Transition Connection Design

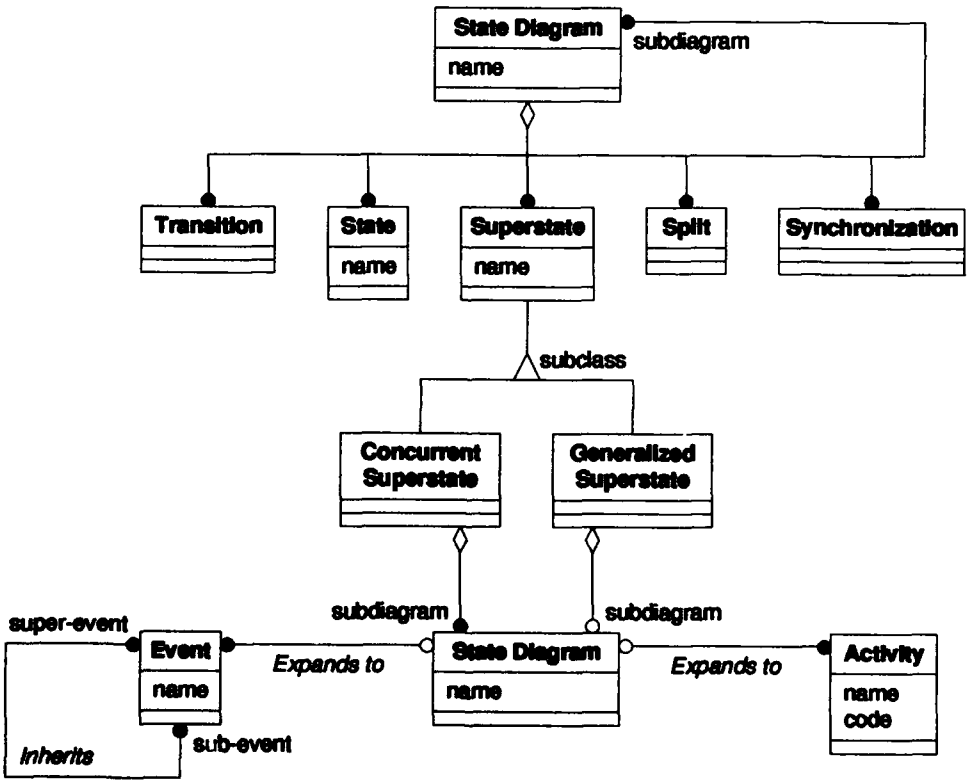


Figure 28. State Diagram Design

Appendix F. *Data Model: Functional Model Design Diagrams*

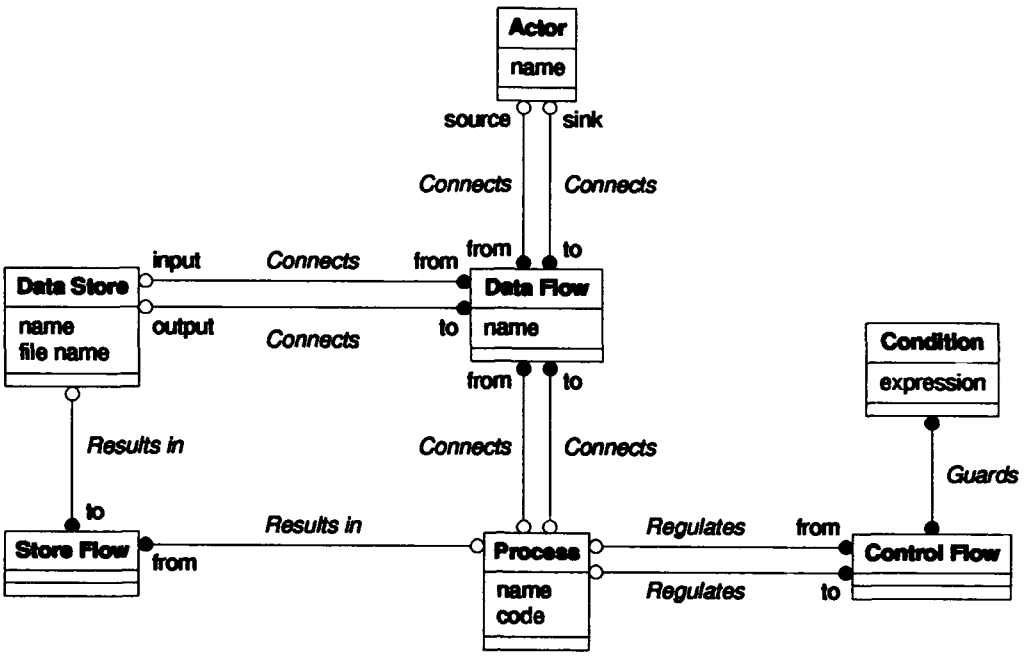


Figure 29. Flow Connection Design

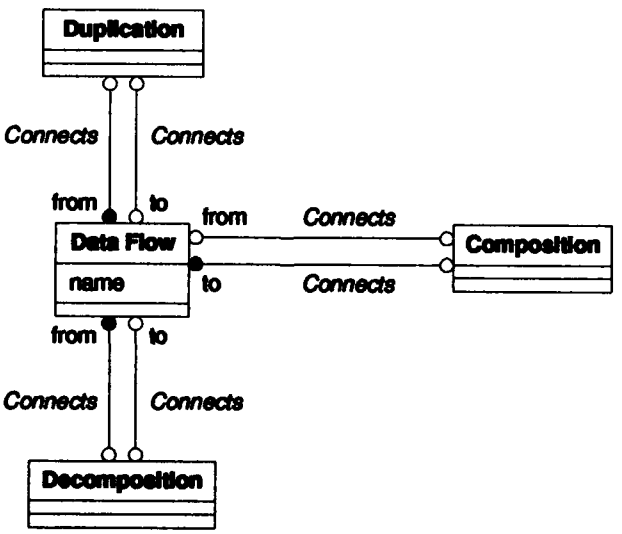


Figure 30. Data Flow Connection Design

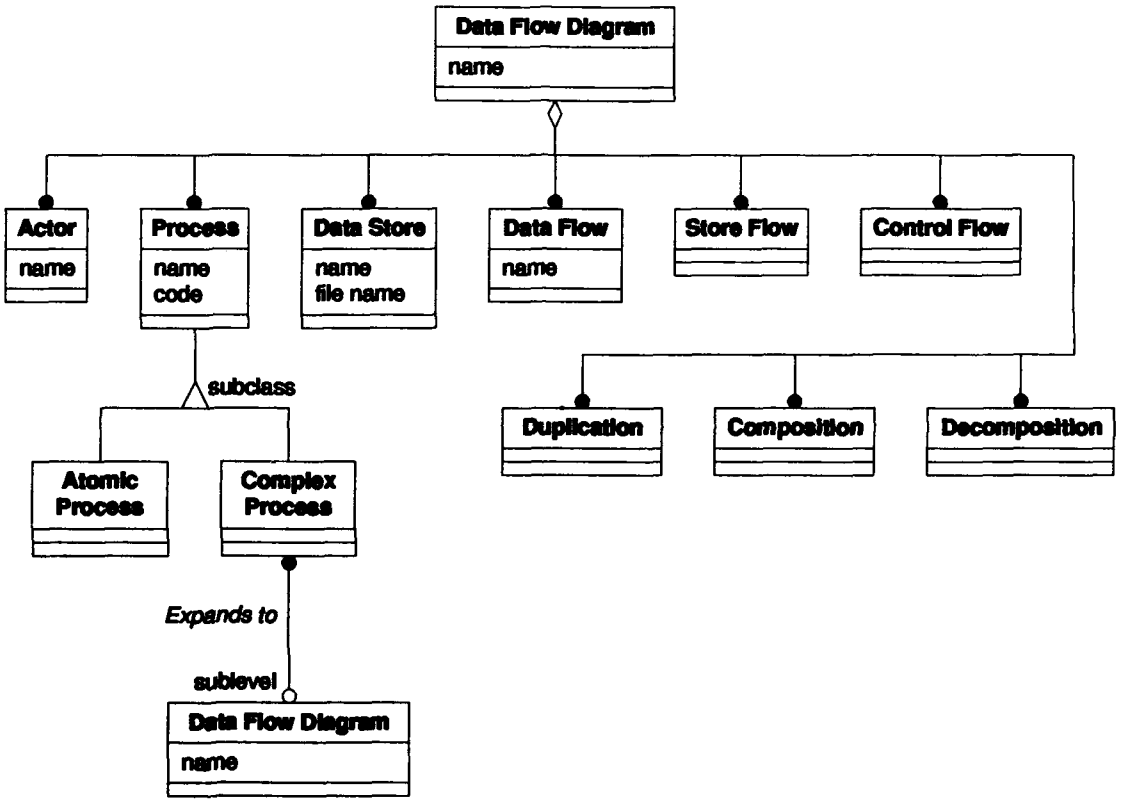


Figure 31. Data Flow Diagram Design

Appendix G. *Data Model: Cross-Links Design Diagrams*

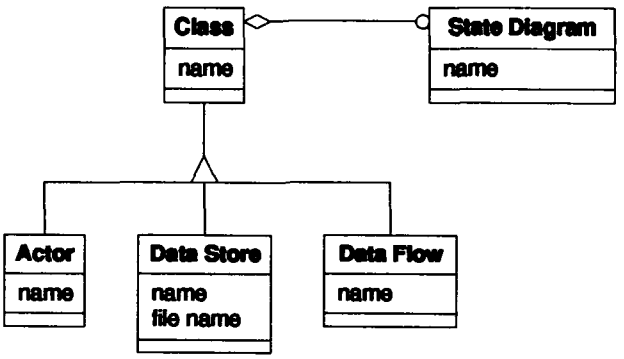


Figure 32. Class Cross-Link Design

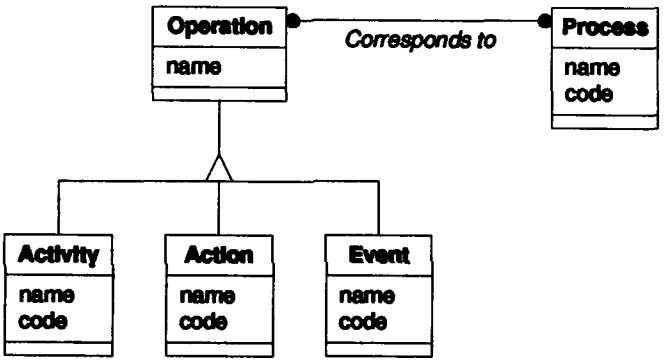


Figure 33. Operation Cross-Link Design

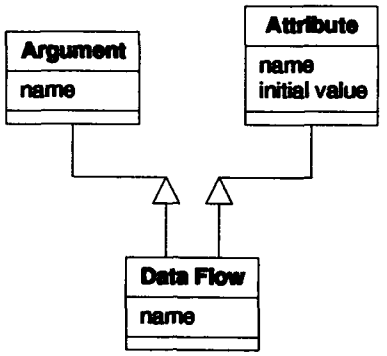


Figure 34. Data Flow Cross-Link Design

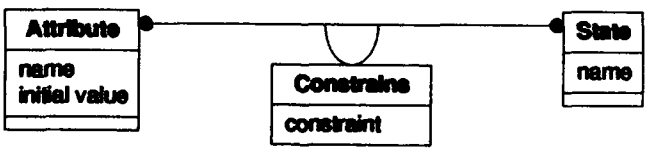


Figure 35. State Cross-Link Design

Appendix H. *Graphical Classes Design Diagram*

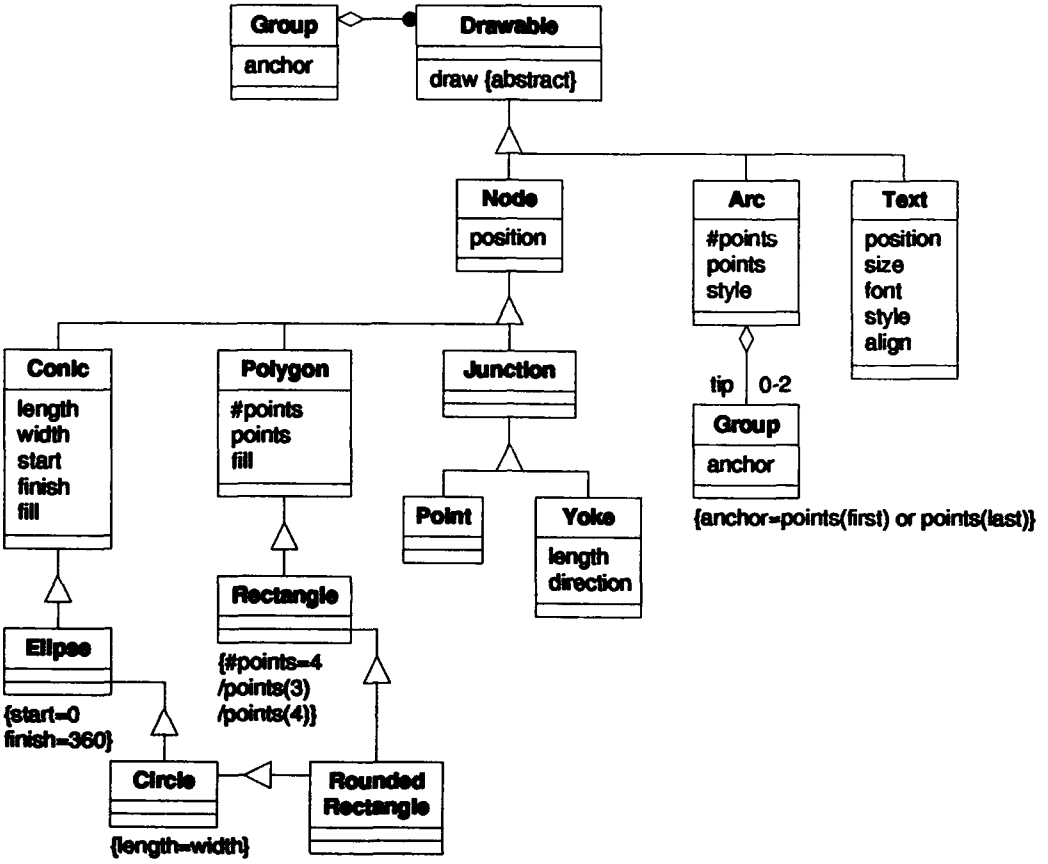


Figure 36. Graphical Classes Design

Appendix I. *Drawing Model Design Diagram*

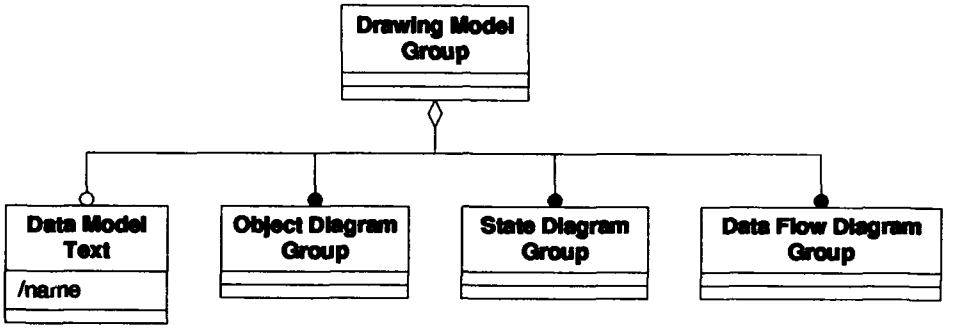


Figure 37. Drawing Model Design

Appendix J. *Drawing Model: Object Model Design Diagrams*

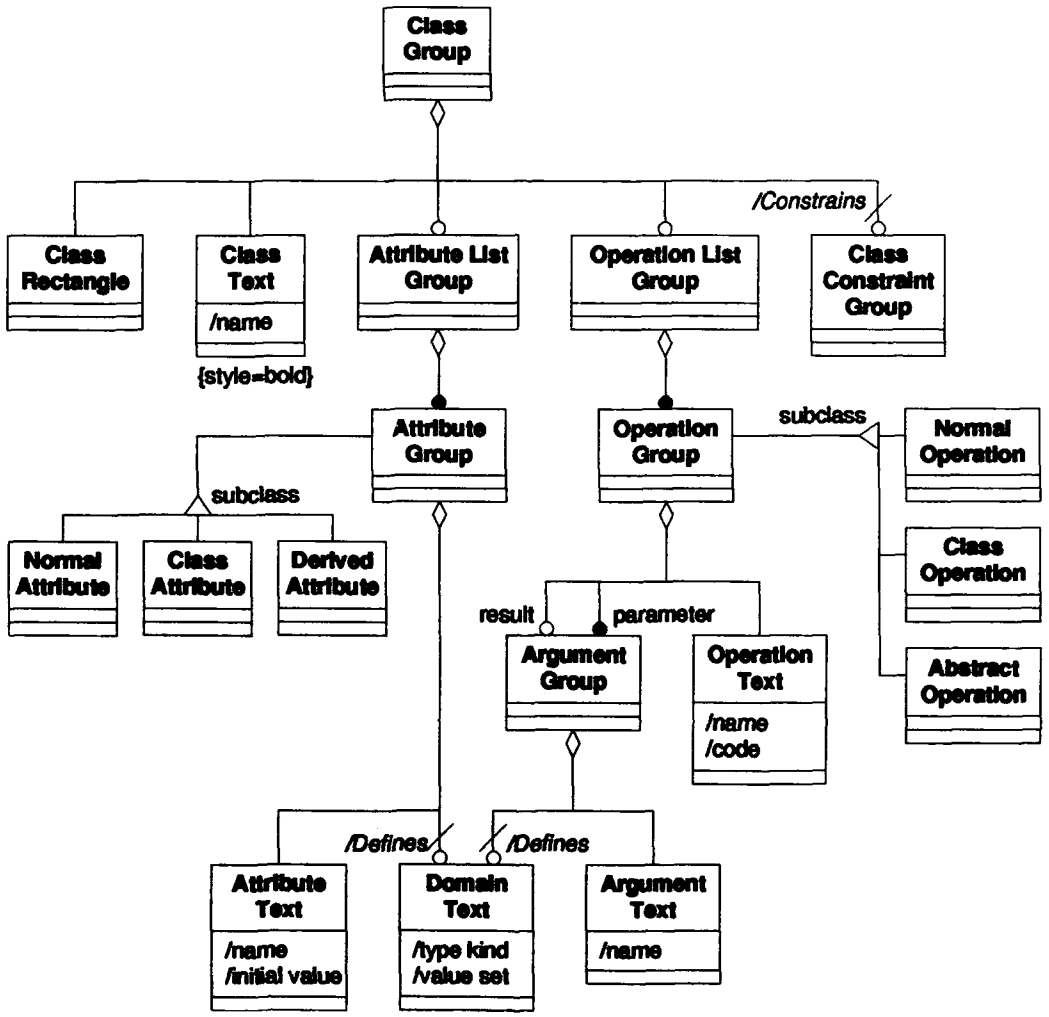


Figure 38. Class Group Design

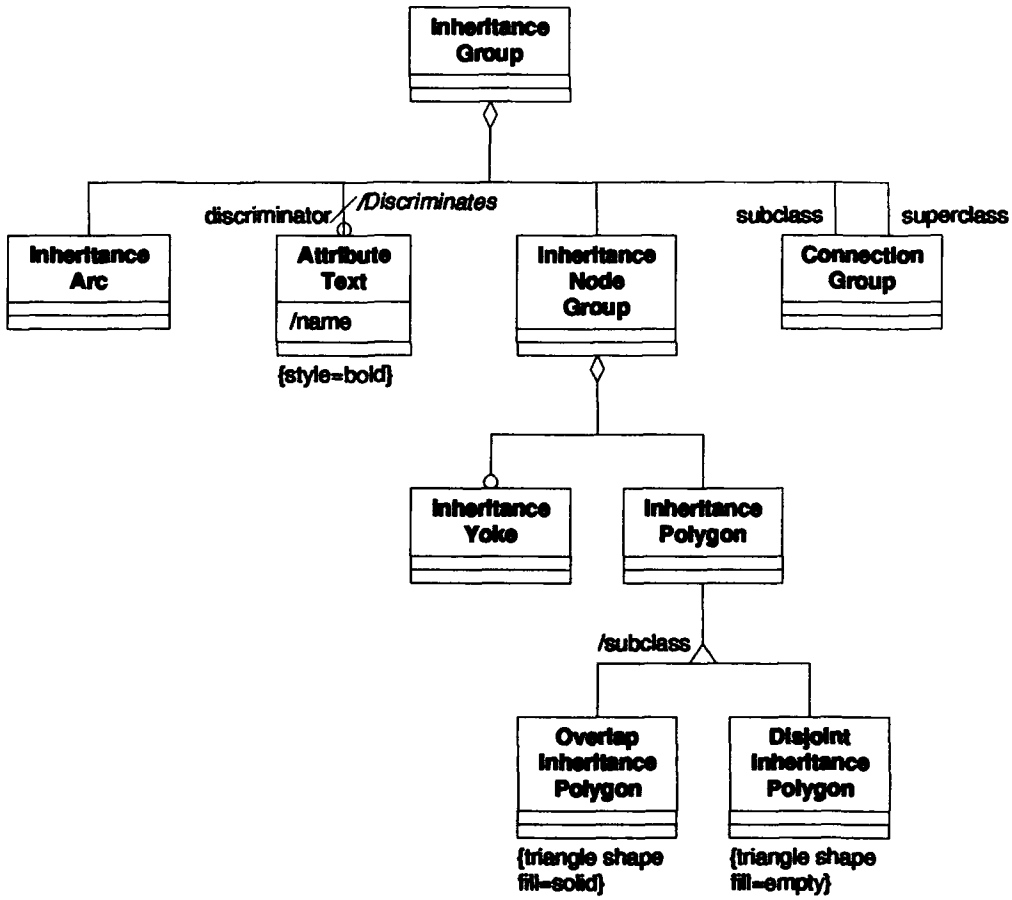


Figure 39. Inheritance Group Design

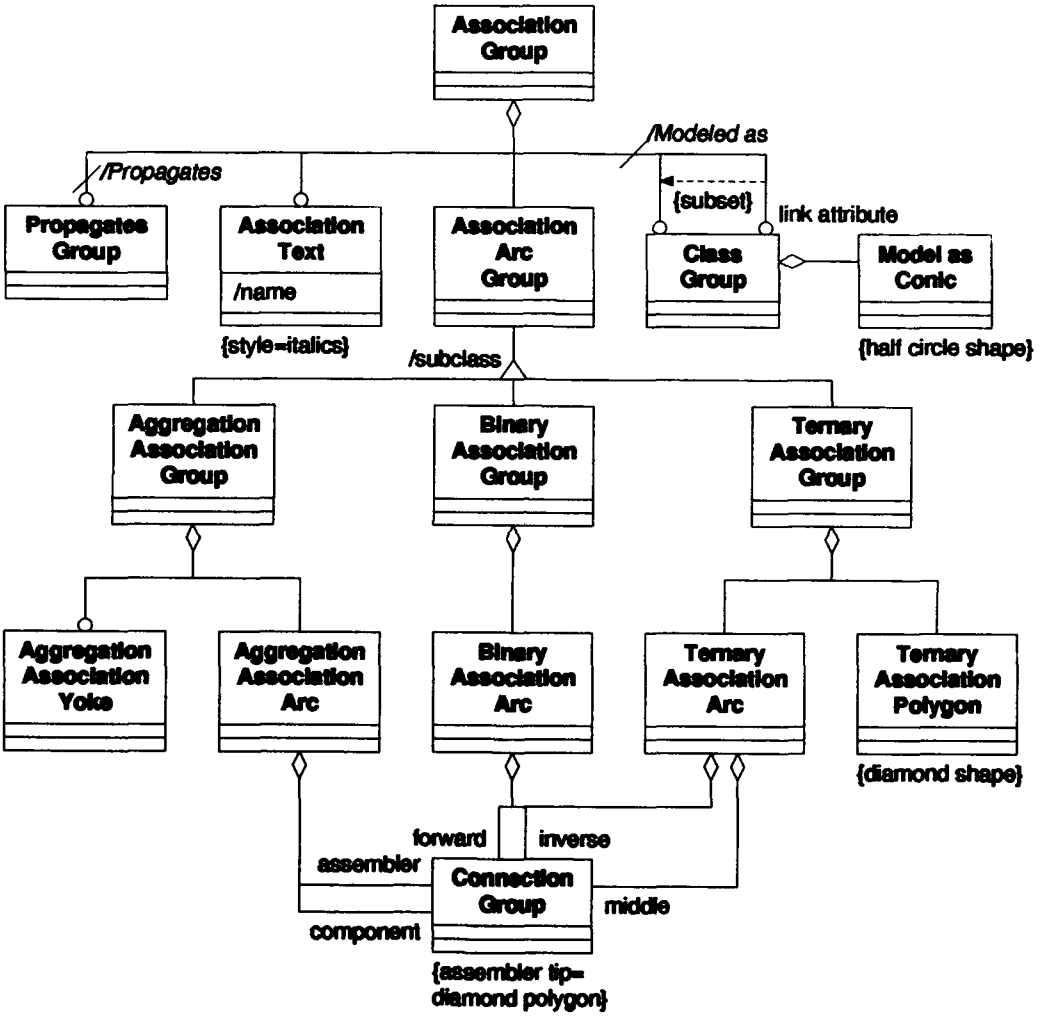


Figure 40. Association Group Design

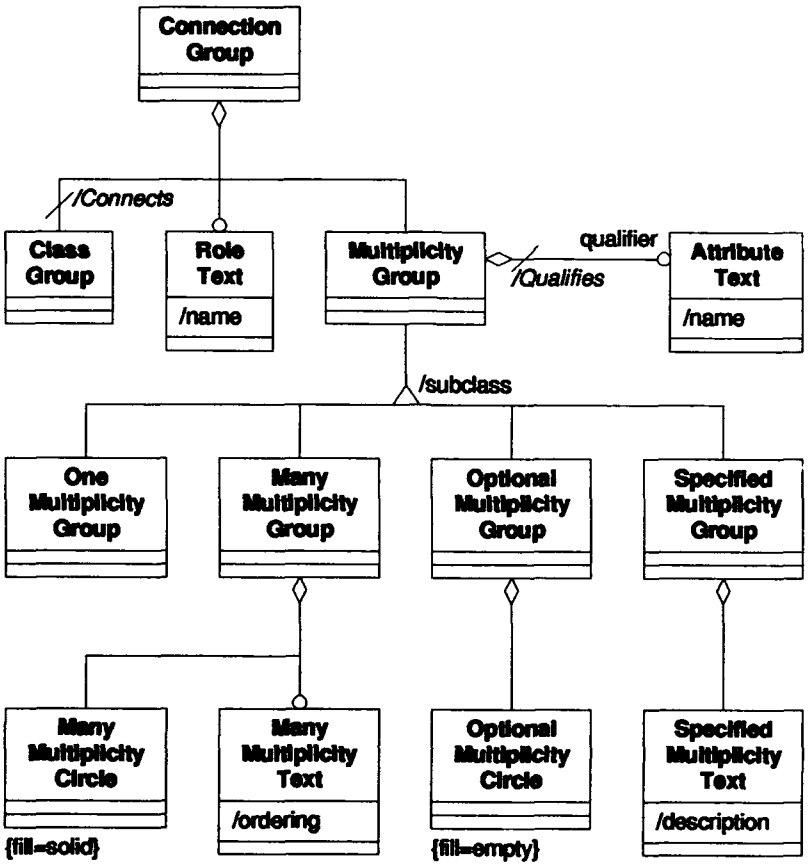


Figure 41. Connection Group Design

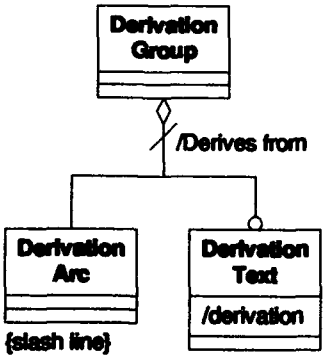


Figure 42. Derivation Group Design

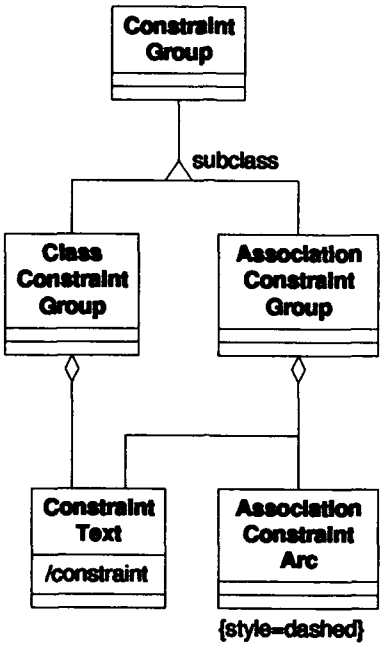


Figure 43. Constraint Group Design

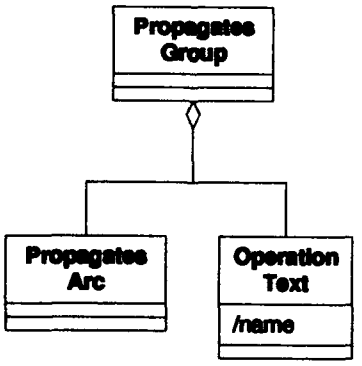


Figure 44. Propagates Group Design

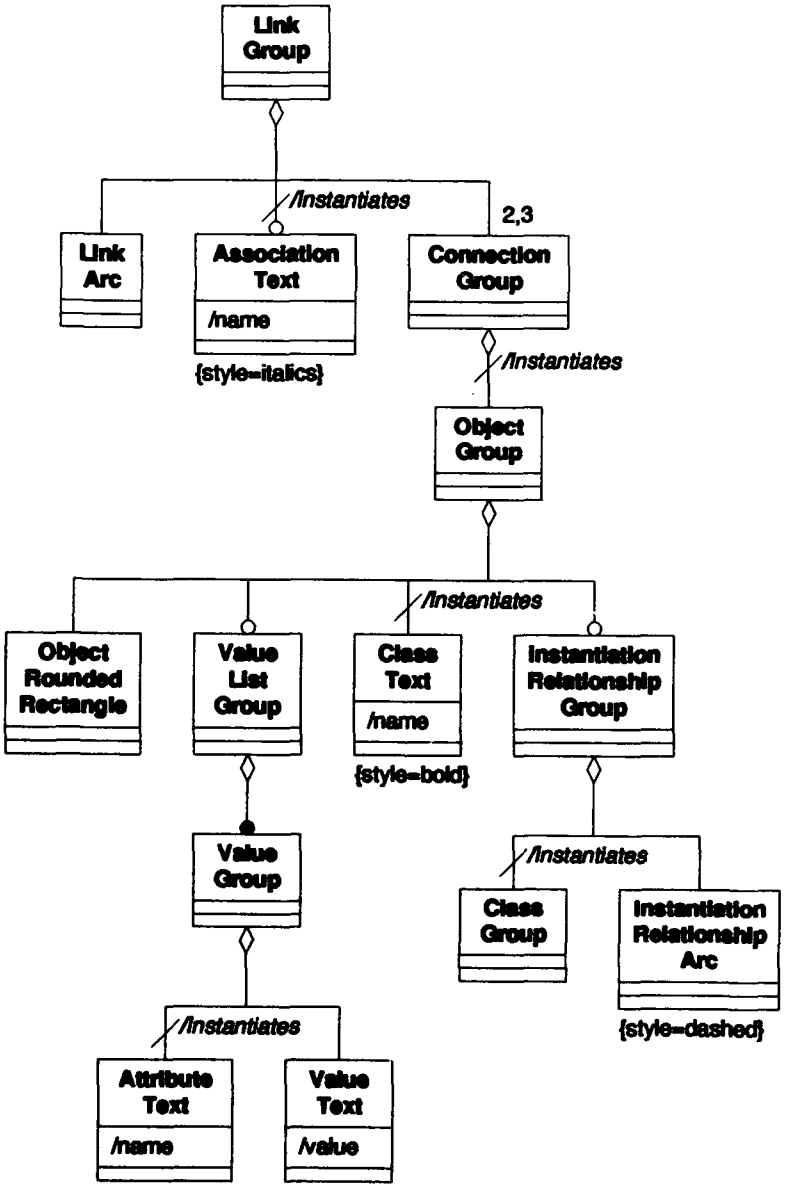


Figure 45. Instantiates Group Design

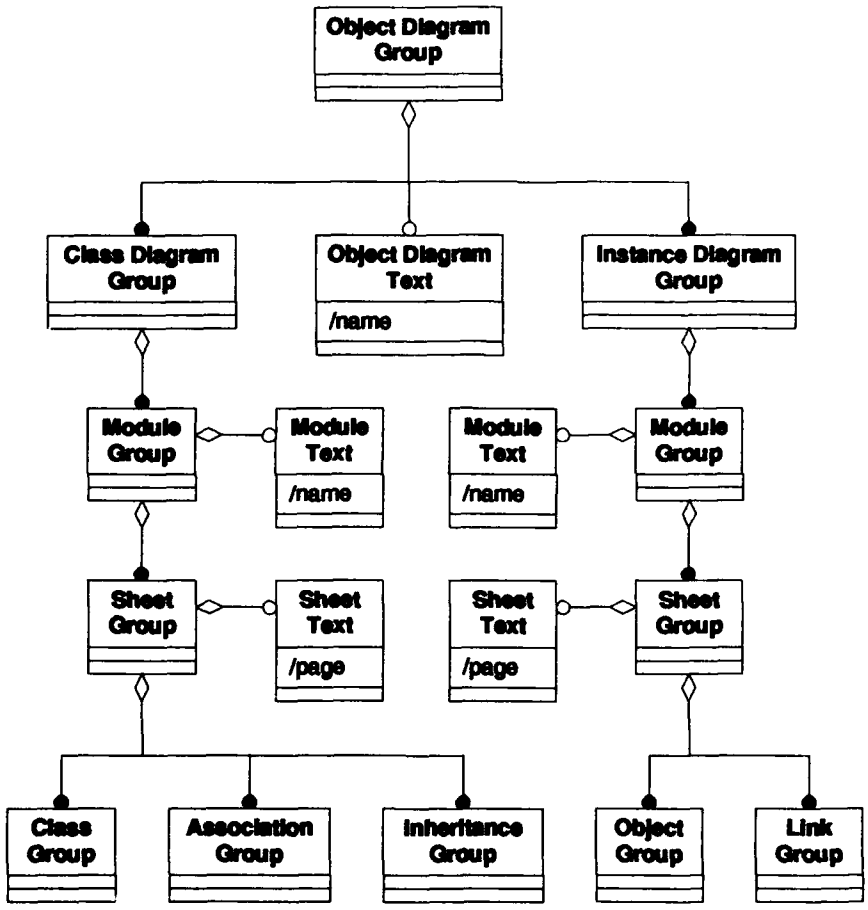


Figure 46. Object Diagram Group Design

Appendix K. *Drawing Model: Dynamic Model Design Diagrams*

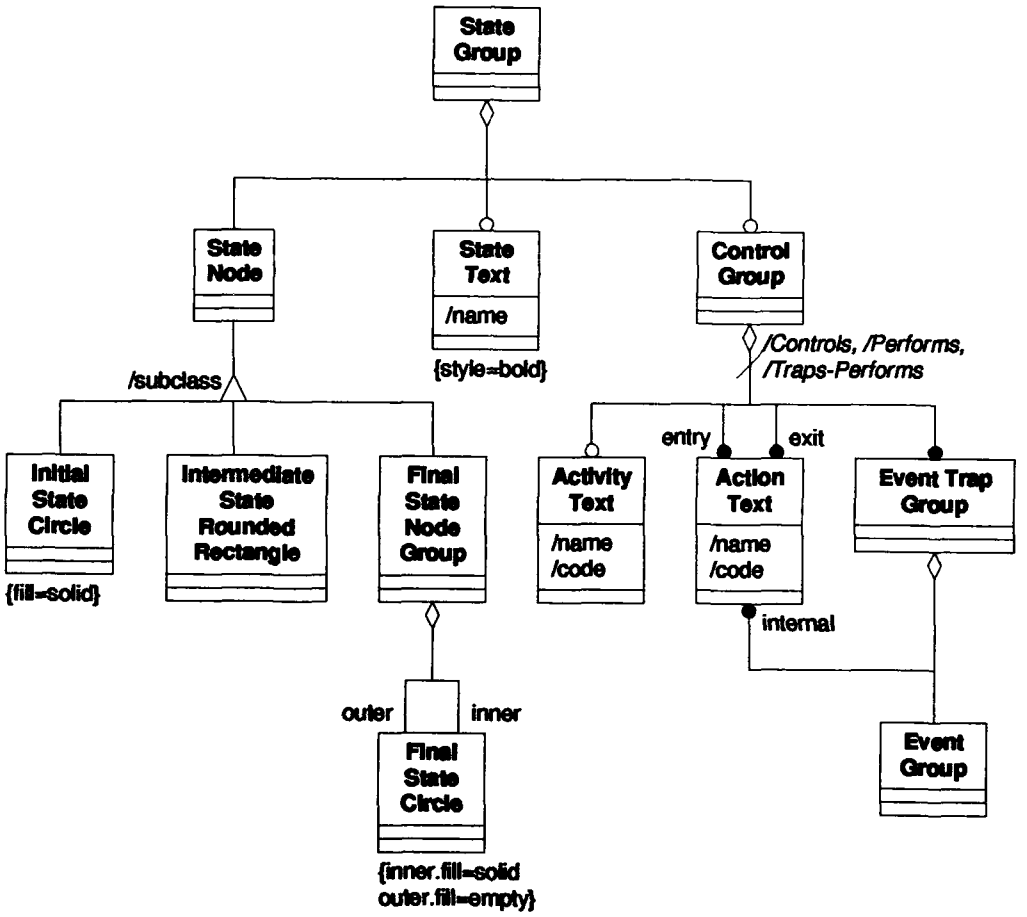


Figure 47. State Group Design

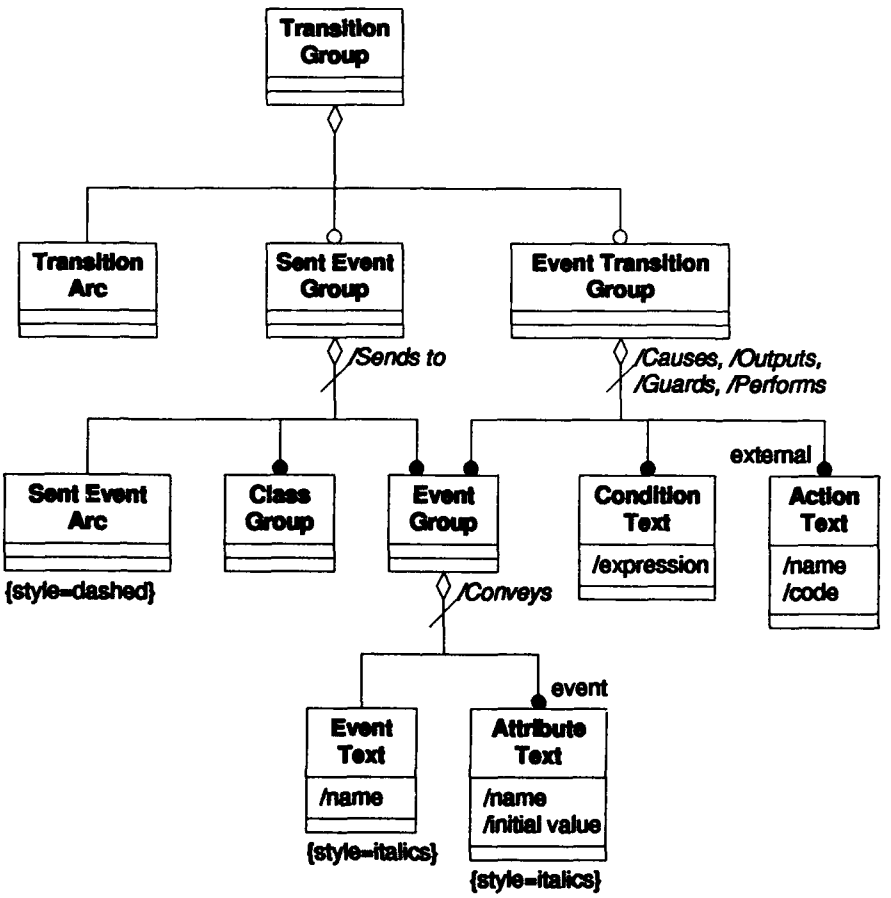


Figure 48. Transition Group Design

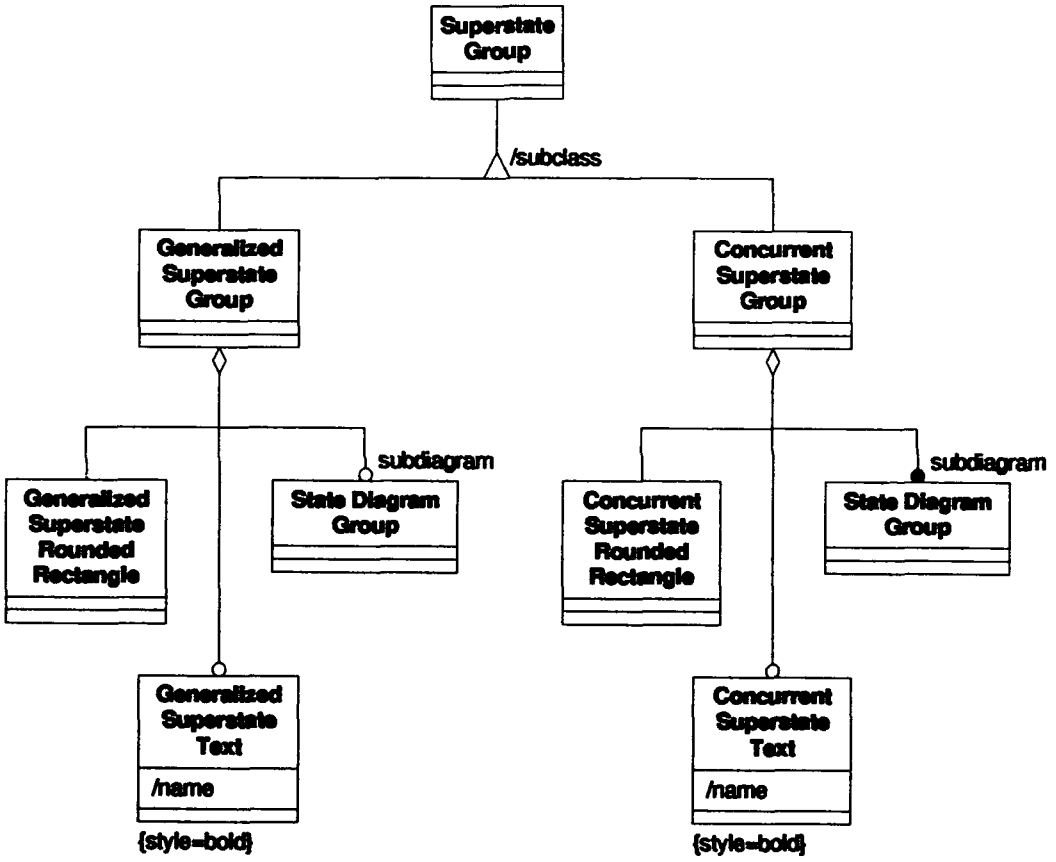


Figure 49. Superstate Group Design

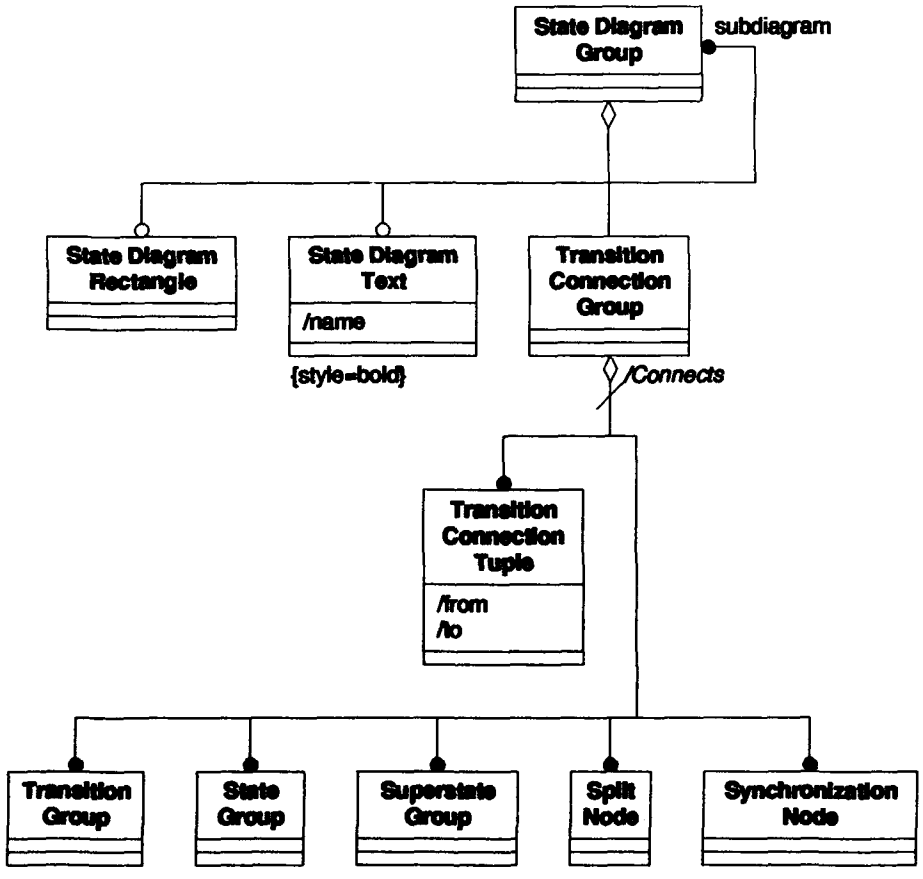


Figure 50. State Diagram Group Design

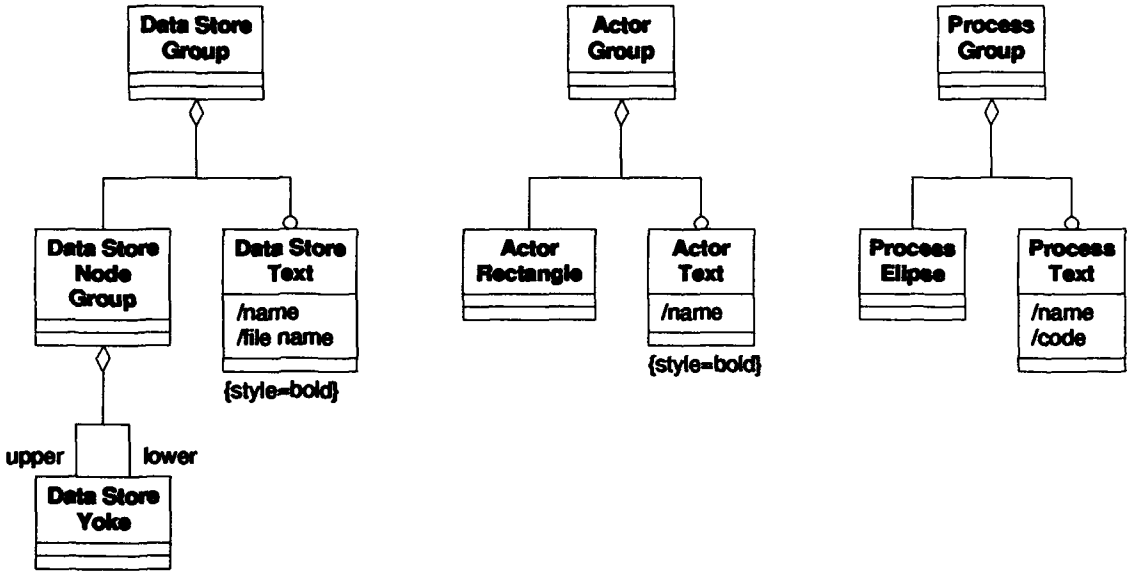


Figure 51. Process, Actor, and Data Store Group Design

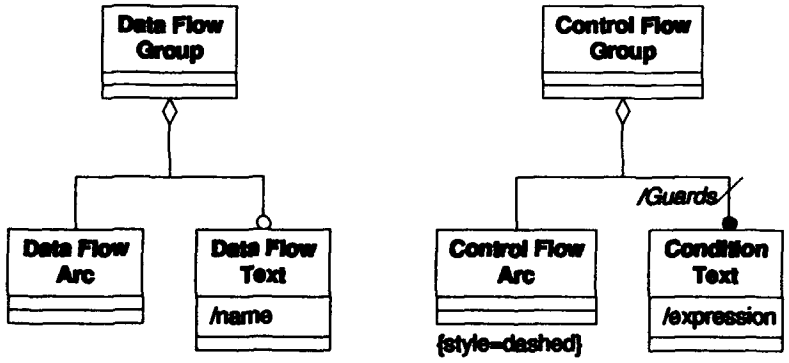


Figure 52. Data Flow and Control Flow Group Design

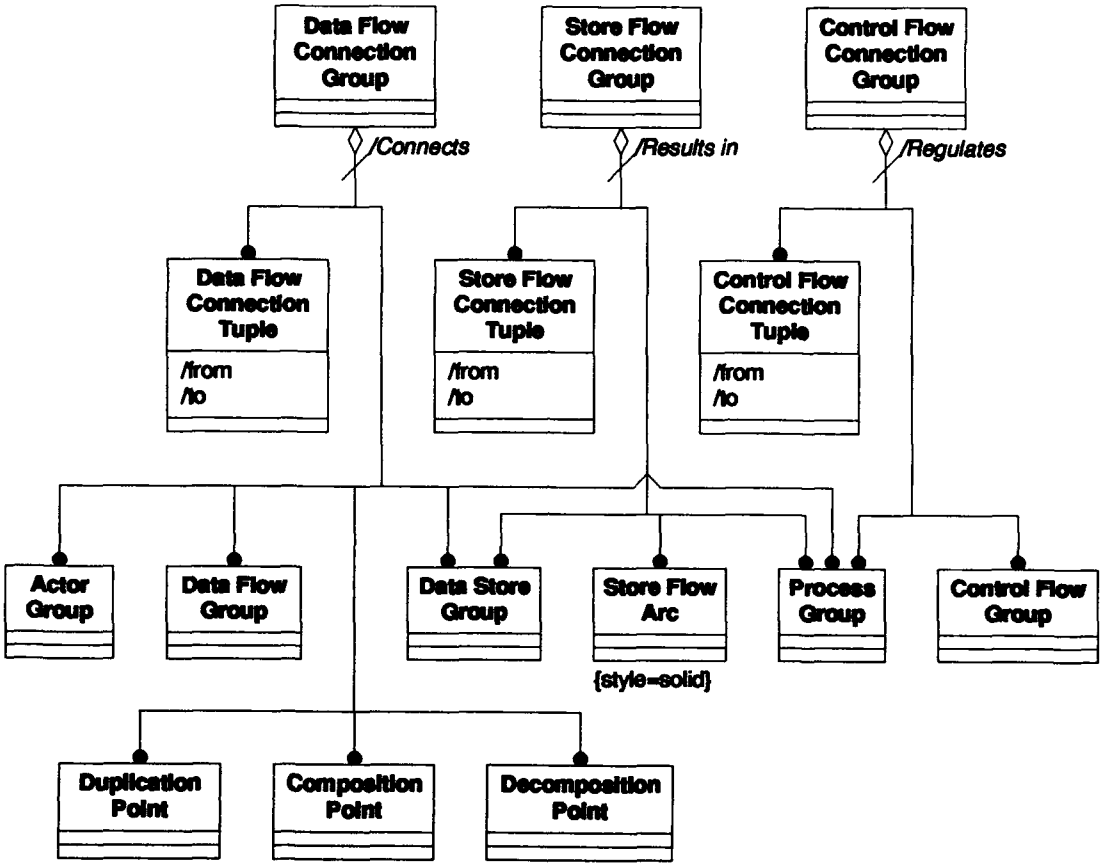


Figure 53. Connection Groups Design

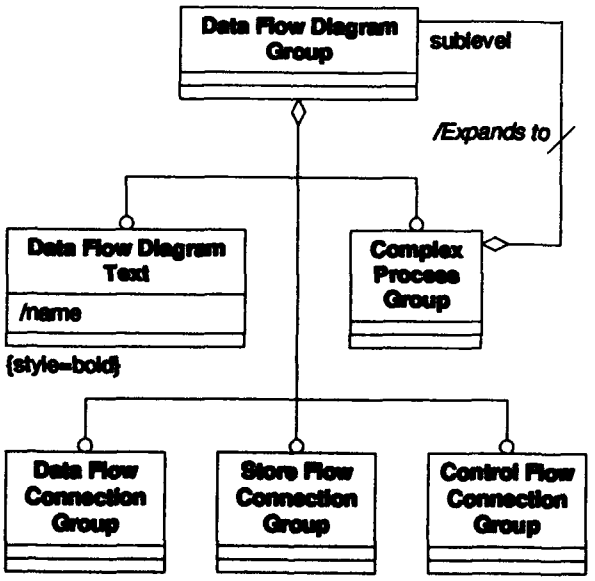


Figure 54. Data Flow Diagram Group Design

Appendix M. *Code Listings*

This appendix contains sample listings of the code developed for this thesis. The following code segments are included:

- *Sequential List Package (specification only)* - shows the code listing for the generic package that is used to instantiate sequential lists. This package is used to generate sequential lists (sorted by object handle) of objects in the object data repository.
- *Indexed Sequential List Package (specification only)* - shows the code listing for the generic package that is used to instantiate indexed-sequential lists. This package is used by the object manager to maintain indexed-sequential lists (sorted and indexed by object handle) of instantiated (created) objects.
- *Object Package* - shows the code listing for the object manager. This listing also contains the generic package that is used to instantiate basic classes.
- *Sample Class Package* - shows the code listing for a sample class. All class packages are of this basic form. Only the class name, attributes, and access operations (*get/set*) are different for each class.
- *Sample Association Package* - shows the code listing for a sample association. All association packages (including aggregations) are of this basic form. Only the association name, attributes, and access operations (*get/set*) are different for each association.

M.1 Sequential List Package

```
-----------------------------------------

generic

  type item_type is private;
  with function "<" (
    left  : item_type;
    right : item_type) return boolean;

package sequential_list_pkg is

  type list_type is private;

  procedure clear (
    list : in out list_type);

  procedure insert (
    item : in     item_type;
    list : in out list_type);

  procedure delete (
    item : in     item_type;
    list : in out list_type);

  procedure get_first (
    item :    out item_type;
    list : in out list_type);

  procedure get_last (
    item :    out item_type;
    list : in out list_type);

  procedure get_previous (
    item :    out item_type;
    list : in out list_type);

  procedure get_next (
    item :    out item_type;
    list : in out list_type);

  procedure load (
    file_name : in     string;
    list      : in out list_type);

  procedure save (
    file_name : in     string;
    list      : in out list_type);

  handle_search_error : exception;
  end_of_list_error   : exception;

private

  type node_type;
  type node_ptr_type is access node_type;
  type node_type is record
    item : item_type;
    prev : node_ptr_type := null;
    next : node_ptr_type := null;
  end record;

  type list_type is record
    first   : node_ptr_type := null;
    last    : node_ptr_type := null;
    current : node_ptr_type := null;
  end record;

end sequential_list_pkg;

-----------------------------------------
```

```
-----------------------------------------

generic

  type key_type is private;
  type item_type is private;
  with function "<" (
    left  : key_type;
    right : key_type) return boolean;

package indexed_sequential_list_pkg is

  type list_type is private;

  procedure rebuild_index (
    list : in out list_type);

  procedure clear (
    list : in out list_type);

  procedure insert (
    key  : in     key_type;
    item : in     item_type;
    list : in out list_type);

  procedure delete (
    key  : in     key_type;
    list : in out list_type);

  procedure get (
    key  : in     key_type;
    item :    out item_type;
    list : in out list_type);

  procedure set (
    key  : in     key_type;
    item : in     item_type;
    list : in out list_type);

  procedure get_first (
    key  :    out key_type;
    item :    out item_type;
    list : in out list_type);

  procedure get_last (
    key  :    out key_type;
    item :    out item_type;
    list : in out list_type);

  procedure get_previous (
    key  :    out key_type;
    item :    out item_type;
    list : in out list_type);

  procedure get_next (
    key  :    out key_type;
    item :    out item_type;
    list : in out list_type);

  procedure load (
    file_name : in     string;
    list      : in out list_type);

  procedure save (
    file_name : in     string;
    list      : in out list_type);

  handle_search_error : exception;
  end_of_list_error   : exception;

private

  type node_type;
  type node_ptr_type is access node_type;
  type node_type is record
    key    : key_type;
    item   : item_type;
    prev   : node_ptr_type := null;
    next   : node_ptr_type := null;
    left   : node_ptr_type := null;
    right  : node_ptr_type := null;
  end record;

  type list_type is record
    first   : node_ptr_type := null;
    last    : node_ptr_type := null;
    current : node_ptr_type := null;
    index   : node_ptr_type := null;
  end record;

end indexed_sequential_list_pkg;

-----------------------------------------
```

```
-------------------------------------------------------------------------------

with sequential_list_pkg;

package object_pkg is

-------------------------------------------------------------------------------

  subtype handle_type is integer;

  null_handle : constant handle_type := 0;

  package handle_list_pkg is new sequential_list_pkg (
    item_type => handle_type,
    "<"       => standard."<");

  type aggregation_type is (
    assembler_object,
    component_object);

  type aggregation_objects_type is
    array (aggregation_type) of handle_type;

  type binary_association_type is (
    forward_object,
    inverse_object);

  type binary_association_objects_type is
    array (binary_association_type) of handle_type;

  type ternary_association_type is (
    forward_object,
    middle_object,
    inverse_object);

  type ternary_association_objects_type is
    array (ternary_association_type) of handle_type;

  procedure initialize_next_handle;

  procedure load_next_handle (
    file_name : in string);

  procedure save_next_handle (
    file_name : in string);

  function get_class_name (
    handle : in handle_type) return string;

  handle_search_error : exception;

-------------------------------------------------------------------------------
```

```
------------------------------------------------------------------------------

  generic

    class_name : in string;
    type attributes_type is private;
    initial_attributes : in attributes_type;

  package basics_pkg is

------------------------------------------------------------------------------

    procedure clear;

    procedure create (
      handle : out handle_type);

    procedure delete (
      handle : in handle_type);

    procedure get (
      handle     : in     handle_type;
      attributes :    out attributes_type);

    procedure set (
      handle     : in handle_type;
      attributes : in attributes_type);

    procedure list (
      handle_list : in out handle_list_pkg.list_type);

    procedure load (
      file_name : in string);

    procedure save (
      file_name : in string);

------------------------------------------------------------------------------

  end basics_pkg;

------------------------------------------------------------------------------
------------------------------------------------------------------------------

end object_pkg;

------------------------------------------------------------------------------
------------------------------------------------------------------------------
------------------------------------------------------------------------------

with label_pkg;
with sequential_io;
with io_exceptions;
with indexed_sequential_list_pkg;

package body object_pkg is
```

```
  package handle_file_pkg is new sequential_io (
    element_type => handle_type);

  next_handle : handle_type;

  next_handle_file_extension : constant string := "next_handle";

  package master_list_pkg is new indexed_sequential_list_pkg (
    key_type  => handle_type,
    item_type => label_pkg.vstring,
    "<"       => standard."<");

  master_list : master_list_pkg.list_type;

  master_list_file_extension : constant string := "master_list";

-------------------------------------------------------------------------------

  procedure allocate_handle (
    handle : out handle_type) is

  begin -- allocate_handle;

    handle := next_handle;

    next_handle := next_handle + 1;

  exception
    when others => raise;

  end allocate_handle;

-------------------------------------------------------------------------------

  procedure deallocate_handle (
    handle : in handle_type) is

  begin -- deallocate_handle;

    null;

  exception
    when others => raise;

  end deallocate_handle;

-------------------------------------------------------------------------------

  procedure initialize_next_handle is

  begin -- initialize_next_handle

    next_handle := 1;
```

```
  exception
    when others => raise;

  end initialize_next_handle;

-------------------------------------------------------------------------------

  procedure load_next_handle (
    file_name : in string) is

  handle_file : handle_file_pkg.file_type;

  begin -- load_next_handle

    handle_file_pkg.open (handle_file, handle_file_pkg.in_file,
      file_name & "." & next_handle_file_extension);

    handle_file_pkg.read (handle_file, next_handle);

    handle_file_pkg.close (handle_file);

  exception
    when others => raise;

  end load_next_handle;

-------------------------------------------------------------------------------

  procedure save_next_handle (
    file_name : in string) is

  handle_file : handle_file_pkg.file_type;

  begin -- save_next_handle

    handle_file_pkg.create (handle_file, handle_file_pkg.out_file,
      file_name & "." & next_handle_file_extension);

    handle_file_pkg.write (handle_file, next_handle);

    handle_file_pkg.close (handle_file);

  exception
    when others => raise;

  end save_next_handle;

-------------------------------------------------------------------------------

  function get_class_name (
    handle : in handle_type) return string is

  class_name : label_pkg.vstring;

  begin -- get_class_name
```

```
    master_list_pkg.get (handle, class_name, master_list);

    return label_pkg.str (class_name);

  exception
    when master_list_pkg.handle_search_error => raise handle_search_error;
    when others => raise;

  end get_class_name;

-------------------------------------------------------------------------------
-------------------------------------------------------------------------------

  package body basics_pkg is

-------------------------------------------------------------------------------

    package object_list_pkg is new indexed_sequential_list_pkg (
      key_type  => handle_type,
      item_type => attributes_type,
      "<"       => standard."<");

    object_list : object_list_pkg.list_type;

-------------------------------------------------------------------------------

    procedure initialize (
      handle : in handle_type) is

    attributes : attributes_type;

    begin -- initialize

      attributes := initial_attributes;

      object_list_pkg.set (handle, attributes, object_list);

    exception
      when others => raise;

    end initialize;

-------------------------------------------------------------------------------

    procedure finalize (
      handle : in handle_type) is

    begin -- finalize

      null;

    exception
      when others => raise;

    end finalize;
```

```
-------------------------------------------------------------------------------

    procedure clear is

    handle     : handle_type;
    attributes : attributes_type;

    begin -- clear

      begin -- trap end_of_list_errors

        object_list_pkg.get_first (handle, attributes, object_list);

        loop
          delete (handle);
          object_list_pkg.get_first (handle, attributes, object_list);
        end loop;

      exception
        when object_list_pkg.end_of_list_error => null;
        when others => raise;

      end; -- trap end_of_list_errors

    exception
      when others => raise;

    end clear;

-------------------------------------------------------------------------------

    procedure create (
      handle : out handle_type) is

    temp_handle : handle_type;
    attributes  : attributes_type;

    begin -- create

      allocate_handle (temp_handle);
      object_list_pkg.insert (temp_handle, attributes, object_list);
      initialize (temp_handle);
      master_list_pkg.insert (
        temp_handle, label_pkg.vstr (class_name), master_list);

      handle := temp_handle;

    exception
      when others => raise;

    end create;

-------------------------------------------------------------------------------

    procedure delete (
```

```
    handle : in handle_type) is

  begin -- delete

    master_list_pkg.delete (handle, master_list);
    finalize (handle);
    object_list_pkg.delete (handle, object_list);
    deallocate_handle (handle);

  exception
    when master_list_pkg.handle_search_error => raise handle_search_error;
    when object_list_pkg.handle_search_error => raise handle_search_error;
    when others => raise;

  end delete;

-------------------------------------------------------------------------------

  procedure get (
    handle     : in     handle_type;
    attributes :    out attributes_type) is

  begin -- get

    object_list_pkg.get (handle, attributes, object_list);

  exception
    when object_list_pkg.handle_search_error => raise handle_search_error;
    when others => raise;

  end get;

-------------------------------------------------------------------------------

  procedure set (
    handle       : in handle_type;
    attributes   : in attributes_type) is

  begin -- set

    object_list_pkg.set (handle, attributes, object_list);

  exception
    when object_list_pkg.handle_search_error => raise handle_search_error;
    when others => raise;

  end set;

-------------------------------------------------------------------------------

  procedure list (
    handle_list : in out handle_list_pkg.list_type) is

  handle     : handle_type;
  attributes : attributes_type;
```

```
  begin -- list

    object_list_pkg.get_first (handle, attributes, object_list);

    loop
      handle_list_pkg.insert (handle, handle_list);
      object_list_pkg.get_next (handle, attributes, object_list);
    end loop;

  exception
    when object_list_pkg.end_of_list_error => null;
    when others => raise;

  end list;

-------------------------------------------------------------------------------

  procedure load (
    file_name : in string) is

  handle     : handle_type;
  attributes : attributes_type;

  begin -- load

    begin -- delete the old handles from the master index list
      object_list_pkg.get_first (handle, attributes, object_list);
      loop
        master_list_pkg.delete (handle, master_list);
        object_list_pkg.get_next (handle, attributes, object_list);
      end loop;
    exception
      when object_list_pkg.end_of_list_error => null;
    end; -- delete the old handles from the master index list

    object_list_pkg.clear (object_list);
    object_list_pkg.load (file_name & "." & class_name, object_list);

    begin -- insert the new handles into the master index list
      object_list_pkg.get_first (handle, attributes, object_list);
      loop
        master_list_pkg.insert (
          handle, label_pkg.vstr (class_name), master_list);
        object_list_pkg.get_next (handle, attributes, object_list);
      end loop;
    exception
      when object_list_pkg.end_of_list_error => null;
    end; -- insert the new handles into the master index list

    master_list_pkg.rebuild_index (master_list);

  exception
    when others => raise;

  end load;
```

```
-------------------------------------------------------------------------------

    procedure save (
      file_name : in string) is

    handle      : handle_type;
    attributes  : attributes_type;
    saved_index : master_list_pkg.list_type;

    begin -- save

      object_list_pkg.save (file_name & "." & class_name, object_list);

      begin -- update the master index list

        master_list_pkg.load (
          file_name & "." & master_list_file_extension, saved_index);

        object_list_pkg.get_first (handle, attributes, object_list);
        loop
          master_list_pkg.insert (
            handle, label_pkg.vstr (class_name), saved_index);
          object_list_pkg.get_next (handle, attributes, object_list);
        end loop;

      exception
        when io_exceptions.name_error =>
          master_list_pkg.save (
            file_name & "." & master_list_file_extension, master_list);
        when object_list_pkg.end_of_list_error =>
          master_list_pkg.save (
            file_name & "." & master_list_file_extension, saved_index);
          master_list_pkg.clear (saved_index);
        when others => raise;

      end; -- update the master index list

    exception
      when others => raise;

    end save;

-------------------------------------------------------------------------------

  end basics_pkg;

-------------------------------------------------------------------------------
-------------------------------------------------------------------------------

end object_pkg;

-------------------------------------------------------------------------------
```

M.4 Class Package

```
-------------------------------------------------------------------------------

with label_pkg;
with object_pkg;

package class is

-------------------------------------------------------------------------------

  class_name : constant string := "class";

  type attributes_type is record
    name     : label_pkg.vstring;
    subclass : label_pkg.vstring;
  end record;

  initial_attributes : constant attributes_type := (
    name     => label_pkg.nul,
    subclass => label_pkg.nul);

  package object_basics_pkg is new object_pkg.basics_pkg (
    class_name         => class_name,
    attributes_type    => attributes_type,
    initial_attributes => initial_attributes);

  procedure get_name (
    handle : in     object_pkg.handle_type;
    name   :    out label_pkg.vstring);

  procedure set_name (
    handle : in object_pkg.handle_type;
    name   : in label_pkg.vstring);

  procedure get_subclass (
    handle   : in     object_pkg.handle_type;
    subclass :    out label_pkg.vstring);

  procedure set_subclass (
    handle   : in object_pkg.handle_type;
    subclass : in label_pkg.vstring);

-------------------------------------------------------------------------------

  procedure clear
  renames object_basics_pkg.clear;

  procedure create (
    handle : out object_pkg.handle_type)
  renames object_basics_pkg.create;

  procedure delete (
    handle : in object_pkg.handle_type)
```

```
  renames object_basics_pkg.delete;

  procedure get (
    handle     : in     object_pkg.handle_type;
    attributes :    out attributes_type)
  renames object_basics_pkg.get;

  procedure set (
    handle     : in object_pkg.handle_type;
    attributes : in attributes_type)
  renames object_basics_pkg.set;

  procedure list (
    handle_list : in out object_pkg.handle_list_pkg.list_type)
  renames object_basics_pkg.list;

  procedure load (
    file_name : in string)
  renames object_basics_pkg.load;

  procedure save (
    file_name : in string)
  renames object_basics_pkg.save;

-------------------------------------------------------------------------------

end class;

-------------------------------------------------------------------------------

package body class is

-------------------------------------------------------------------------------

  procedure get_name (
    handle : in     object_pkg.handle_type;
    name   :    out label_pkg.vstring) is

  attributes : attributes_type;

  begin -- get_name

    get (handle, attributes);

    name := attributes.name;

  exception
    when others => raise;

  end get_name;

-------------------------------------------------------------------------------

  procedure set_name (
    handle : in object_pkg.handle_type;
    name   : in label_pkg.vstring) is
```

```
  attributes : attributes_type;

  begin -- set_name

    get (handle, attributes);

    attributes.name := name;

    set (handle, attributes);

  exception
    when others => raise;

  end set_name;

-------------------------------------------------------------------------------

  procedure get_subclass (
    handle   : in     object_pkg.handle_type;
    subclass :    out label_pkg.vstring) is

  attributes : attributes_type;

  begin -- get_subclass

    get (handle, attributes);

    subclass := attributes.subclass;

  exception
    when others => raise;

  end get_subclass;

-------------------------------------------------------------------------------

  procedure set_subclass (
    handle   : in object_pkg.handle_type;
    subclass : in label_pkg.vstring) is

  attributes : attributes_type;

  begin -- set_subclass

    get (handle, attributes);

    attributes.subclass := subclass;

    set (handle, attributes);

  exception
    when others => raise;

  end set_subclass;
```

```
--------------------------------------------------------------------------

end class;

--------------------------------------------------------------------------
```

M.5 Association Package

```
-------------------------------------------------------------------------------

with label_pkg;
with object_pkg;

package association is

-------------------------------------------------------------------------------

  association_name : constant string := "association";

  type attributes_type is record
    objects : object_pkg.binary_association_objects_type;
    role    : label_pkg.vstring;
  end record;

  initial_attributes : constant attributes_type := (
    objects => (others => object_pkg.null_handle),
    role    => label_pkg.nul);

  package object_basics_pkg is new object_pkg.basics_pkg (
    class_name         => association_name,
    attributes_type    => attributes_type,
    initial_attributes => initial_attributes);

  procedure get_objects (
    handle  : in     object_pkg.handle_type;
    objects :    out object_pkg.binary_association_objects_type);

  procedure set_objects (
    handle  : in object_pkg.handle_type;
    objects : in object_pkg.binary_association_objects_type);

  procedure get_role (
    handle : in     object_pkg.handle_type;
    role   :    out label_pkg.vstring);

  procedure set_role (
    handle : in object_pkg.handle_type;
    role   : in label_pkg.vstring);

-------------------------------------------------------------------------------

  procedure clear
  renames object_basics_pkg.clear;

  procedure create (
    handle : out object_pkg.handle_type)
  renames object_basics_pkg.create;

  procedure delete (
    handle : in object_pkg.handle_type)
```

```
  renames object_basics_pkg.delete;

  procedure get (
    handle     : in     object_pkg.handle_type;
    attributes :    out attributes_type)
  renames object_basics_pkg.get;

  procedure set (
    handle     : in object_pkg.handle_type;
    attributes : in attributes_type)
  renames object_basics_pkg.set;

  procedure list (
    handle_list : in out object_pkg.handle_list_pkg.list_type)
  renames object_basics_pkg.list;

  procedure load (
    file_name : in string)
  renames object_basics_pkg.load;

  procedure save (
    file_name : in string)
  renames object_basics_pkg.save;

-------------------------------------------------------------------------------

end association;

-------------------------------------------------------------------------------

package body association is

-------------------------------------------------------------------------------

  procedure get_objects (
    handle  : in     object_pkg.handle_type;
    objects :    out object_pkg.binary_association_objects_type) is

  attributes : attributes_type;

  begin -- get_objects

    get (handle, attributes);

    objects := attributes.objects;

  exception
    when others => raise;

  end get_objects;

-------------------------------------------------------------------------------

  procedure set_objects (
    handle  : in object_pkg.handle_type;
    objects : in object_pkg.binary_association_objects_type) is
```

```
  attributes : attributes_type;

  begin -- set_objects

    get (handle, attributes);

    attributes.objects := objects;

    set (handle, attributes);

  exception
    when others => raise;

  end set_objects;

-------------------------------------------------------------------------------

  procedure get_role (
    handle : in     object_pkg.handle_type;
    role   :    out object_pkg.binary_association_objects_type) is

  attributes : attributes_type;

  begin -- get_role

    get (handle, attributes);

    role := attributes.role;

  exception
    when others => raise;

  end get_role;

-------------------------------------------------------------------------------

  procedure set_role (
    handle : in object_pkg.handle_type;
    role   : in object_pkg.binary_association_objects_type) is

  attributes : attributes_type;

  begin -- set_role

    get (handle, attributes);

    attributes.role := role;

    set (handle, attributes);

  exception
    when others => raise;

  end set_role;
```

```
-----------------------------------------------------------------------

end association;

-----------------------------------------------------------------------
```

Appendix N. *Requirements and Design Validation*

This appendix contains a consolidated list of the data and drawing elements abstracted from the OMT. This appendix also contains a sample set of instance diagrams to show how the data and drawing models adequately capture the respective elements of the OMT.

N.1 List of Essential Data Elements

The essential data elements of the OMT are listed below. These elements were extracted from the requirements analysis of the OMT. Each element was designed and implemented directly as a class, association, or aggregation. Attributes for each element are listed in parenthesis.

- Corresponds-to Association (Process—Operation)
- Data Model Class (name)
- Data Model Aggregation (Object Diagram, State Diagram, Data Flow Diagram)
- Object Model Elements

1. Argument Class (name)
2. Association Aggregation (role, Connection)
3. Association Class (name, subclass)
4. Attribute Class (name, initial value)
5. Behavior Class (code)
6. Class Aggregation (Attribute, Operation)
7. Class Class (name)
8. Connection Aggregation (Role, Multiplicity)
9. Connection Class
10. Connects-connection Association (Class—Connection)
11. Constrains Association (constraint, Class—Attribute / Association—Association / State—Attribute)
12. Defines Association (Argument—Domain / Attribute—Domain)
13. Derives-from Association (derivation, Class/Association/Attribute—same)
14. Discriminates Association (Inheritance—Attribute)
15. Domain Class (type, value set)
16. Inheritance Aggregation (Connection)

17. Inheritance Class
18. Instantiates Association (Association—Link / Attribute—Value / Class—Object)
19. Link Aggregation (Connection)
20. Link Class (id)
21. Modeled-as Association (role, Association—Class)
22. Module Aggregation (Sheet)
23. Module Class (name)
24. Multiplicity Class (subclass, ordering, description)
25. Object Aggregation (Value)
26. Object Class (id)
27. Object Diagram Aggregation (Class Diagram, Instance Diagram)
28. Object Diagram Class (name)
29. Operation Aggregation (Signature, Behavior)
30. Operation Class (name, subclass)
31. Propagates Association (Association—Operation)
32. Qualifies Association (Attribute—Multiplicity)
33. Role Class (name)
34. Sheet Aggregation (Class, Association, Inheritance, Object, Link)
35. Sheet Class (page)
36. Signature Aggregation (role, Argument)
37. Signature Class
38. Value Class (id, value)

- Dynamic Model Elements

1. Action Class (name, code)
2. Activity Class (name, code)
3. Causes Association (Event—Transition)
4. Condition Class (expression)
5. Connects Transition Association (role, State/Superstate/Split/Synchronization—same)
6. Controls Association (State—Activity)
7. Conveys Association (Event—Attribute)
8. Event Class (name)
9. Expands-to State Diagram Association (Event—State Diagram / Activity—State Diagram)
10. Generates Association (Activity—Event / Action—Event)
11. Guards Association (Condition—Transition)
12. Inherits Event Association (role, Event)

13. Outputs Association (Transition—Event)
14. Performs Association (role, State—Action)
15. Sends-to Association (Transition—Event—Class)
16. Split Class
17. State Class (name, subclass)
18. State Diagram Aggregation (Transition, State, Superstate, Split, Synchronization, State Subdiagram)
19. State Diagram Class (name)
20. Superstate Aggregation (State Subdiagram)
21. Superstate Class (name, subclass)
22. Synchronization Class
23. Transition Class
24. Traps-Performs Association (State—Event—Action)

- Functional Model Elements

1. Actor Class (name)
2. Composition Class
3. Connects Data Flow Association (role, Actor / Data Store / Data Flow / Process / Duplication / Composition / Decomposition—same)
4. Control Flow Class
5. Data Flow Class (name)
6. Data Flow Diagram Aggregation (Actor, Process, Data Store, Data Flow, Store Flow, Control Flow, Duplication, Composition, Decomposition)
7. Data Flow Diagram Class (name)
8. Data Store Class (name, file name)
9. Decomposition Class
10. Duplication Class
11. Expands-to DFD Association (Process—Data Flow Diagram)
12. Process Class (name, code)
13. Regulates Association (role, Control Flow—Process)
14. Results-in Association (Process—Store Flow / Store Flow—Data Store)
15. Store Flow Class

N.2 List of Drawing Elements

The drawing elements of the OMT are listed below. Each class in the drawing model design diagrams is one of these elements.

1. Arc Aggregation (Group)
2. Arc Class (no. points, points, style)
3. Circle Class - (position, length, width, start, finish, fill)
4. Conic Class - (position, length, width, start, finish, fill)
5. Drawable Class - abstract class
6. Elipse Class - (position, length, width, start, finish, fill)
7. Group Aggregation (Drawable)
8. Group Class (anchor)
9. Junction Class (position) abstract class
10. Node Class - (position) abstract class
11. Point Class (position)
12. Polygon Class (position, no. points, points, fill)
13. Rectangle Class (position, no. points, points, fill)
14. Rounded Rectangle Class (no. points, points, position, length, width, start, finish, fill)
15. Text Class (position, size, font, style, align)
16. Yoke Class (position, length, direction)

N.3 Sample Instance Diagrams

To validate that the data and drawing models developed for this thesis adequately capture the data and drawing elements of the OMT, several sample OMT diagrams were taken from Rumbaugh's text (17) and instantiated. This section contains some of these instance diagrams. Additional example instantiations were done by this author on scrap paper and are not included in this report. Every sample that was tested was able to be instantiated with the developed data and drawing models.

N.3.1 Windowing System Instance Diagram Portions of the object model of the windowing system found in Rumbaugh's text are instantiated in this section (17:44). Figure 55 shows the object model of this windowing system. Figure 56 shows partial instantiations of the windowing system. These diagrams show examples of grouping constructs, inheritance, association, role, multiplicity, and class structure. The following data model design modules are used:

- Data Model Design - see Figure 15

- Object Diagram Design - see Figure 24
- Class Design - see Figure 16
- Inheritance Design - see Figure 17
- Association Design - see Figure 18
- Connection Design - see Figure 19

Figure 57 contains additional instance diagrams from portions of the windowing system. These diagrams show examples of an association constraint, a qualifier, multiple inheritance, an aggregation, roles, and ordering. The following data model design modules are used:

- Connection Design - see Figure 19
- Association Design - see Figure 18
- Association Constraint Design - see Figure 21

Because of the size of this example (about 120 data and 160 drawing elements), only part of the windowing system was instantiated. The figures contain representative segments of the complete instance diagram.

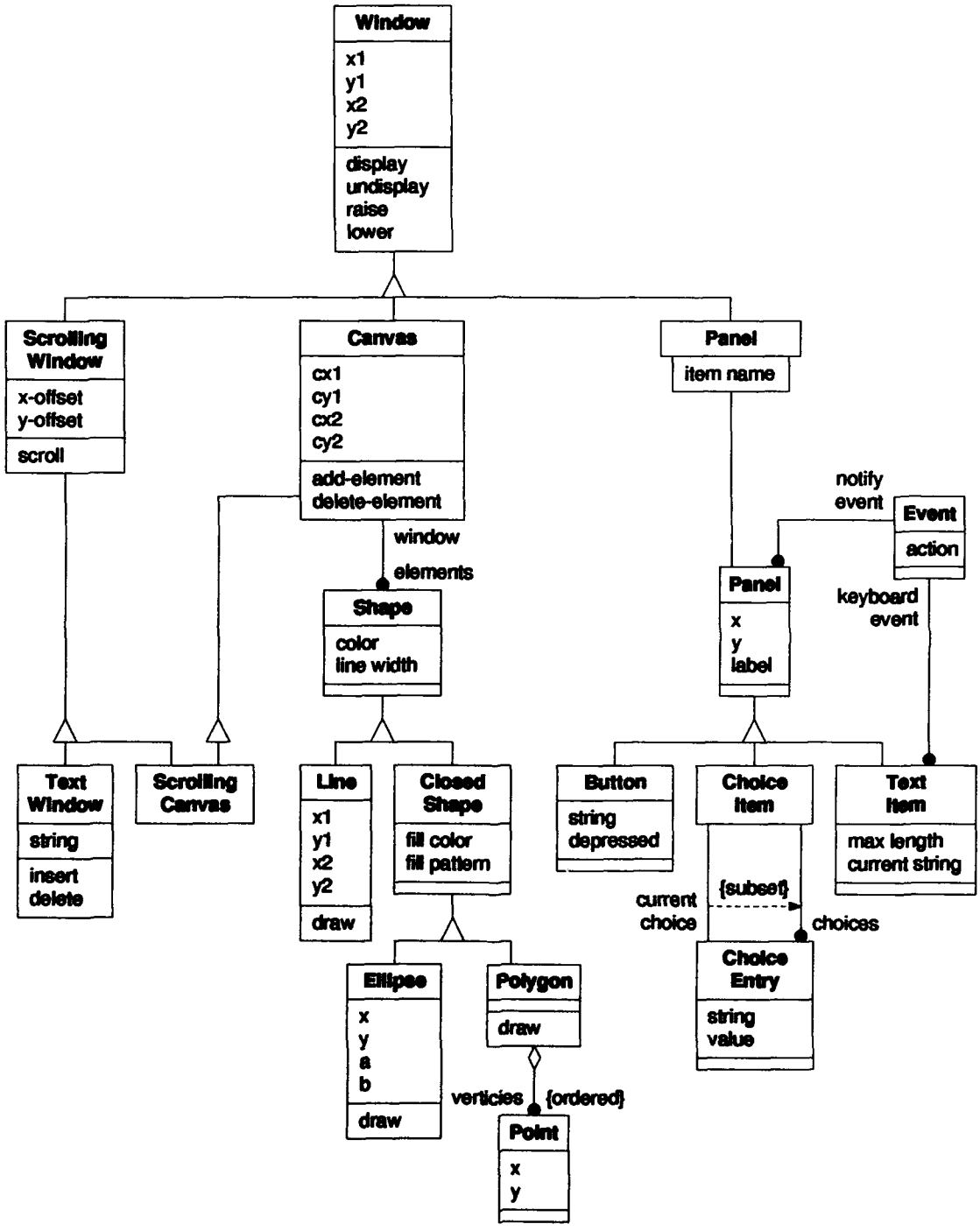


Figure 55. Windowing System Object Diagram

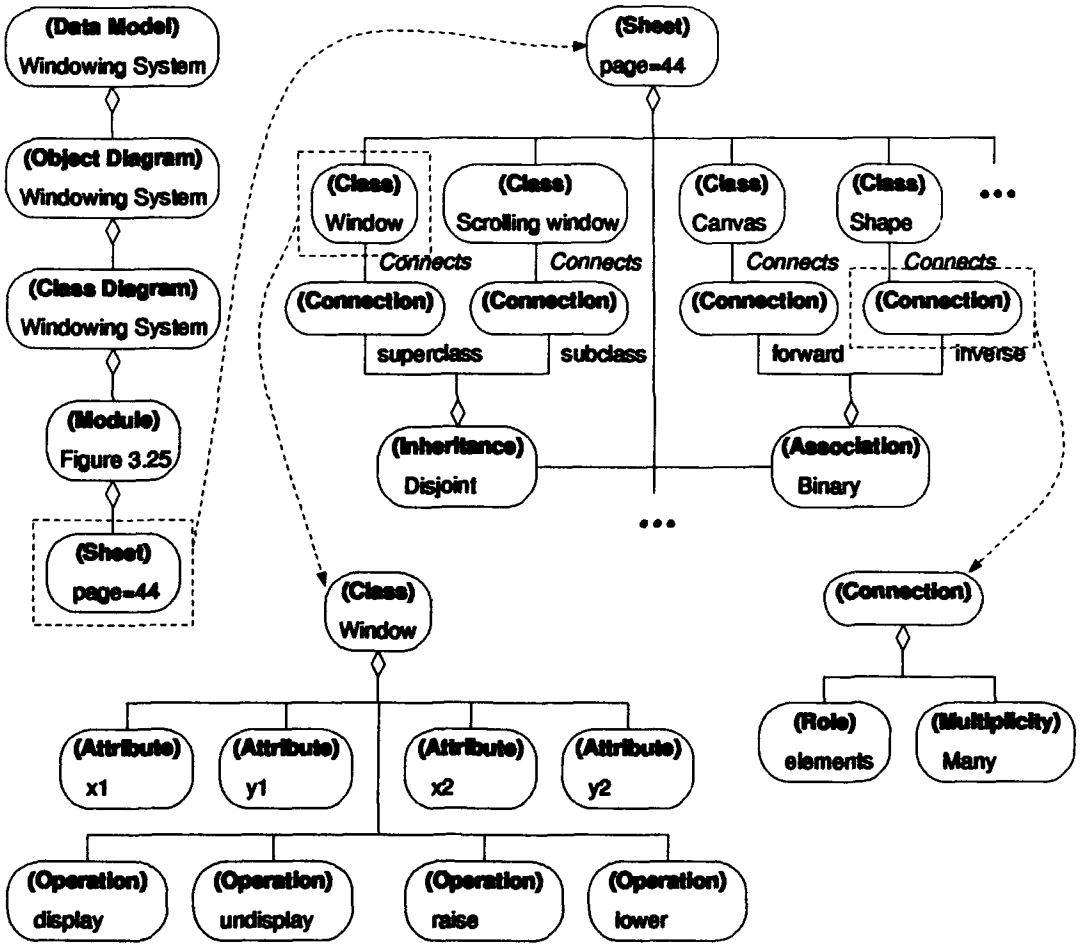


Figure 56. Windowing System Instance Diagram No.1

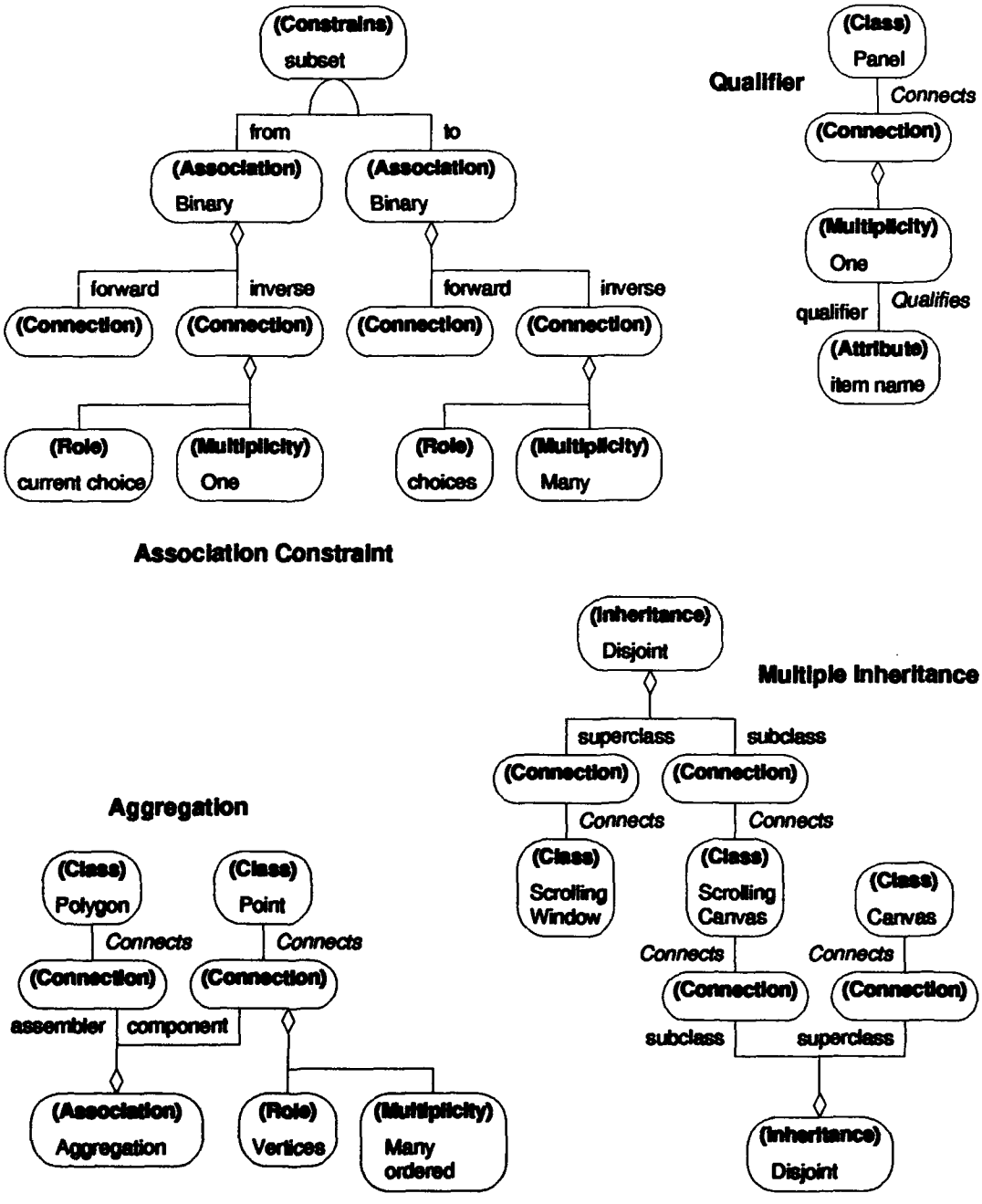


Figure 57. Windowing System Instance Diagram No.2

N.3.2 Car System Instance Diagram The dynamic model of the car system found in Rumbaugh's text are instantiated in this section (17:100). Figure 58 shows the object and state-transition models of this car system. Figures 59–64 show the full instantiation of the car system. These diagrams show examples of states, transitions, events, state generalization, guard conditions, and the cross-link between classes and state diagrams (each class has its own state diagram). The following data model design modules are used:

- Data Model Design - see Figure 15
- Class Cross-Link Design - see Figure 32
- State Design - see Figure 25
- Transition Design - see Figure 26
- State-Transition Connection Design - see Figure 27
- State Diagram Design - see Figure 28

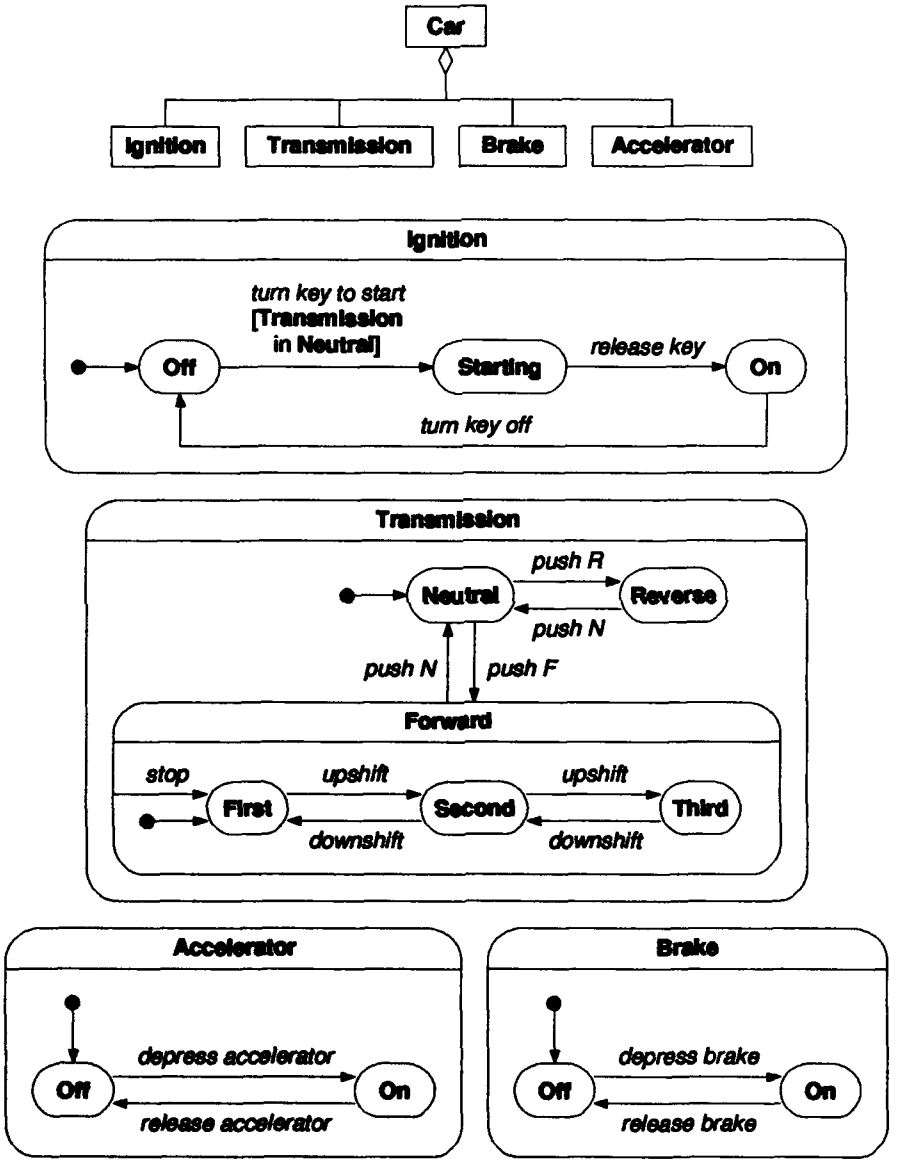


Figure 58. Car System Object and State-Transition Diagrams

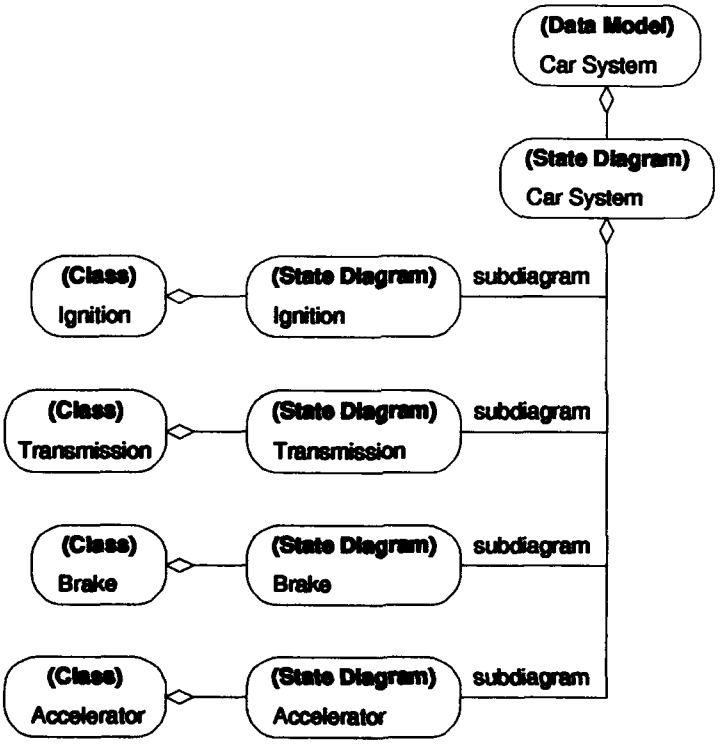


Figure 59. Car System Instance Diagram No.1

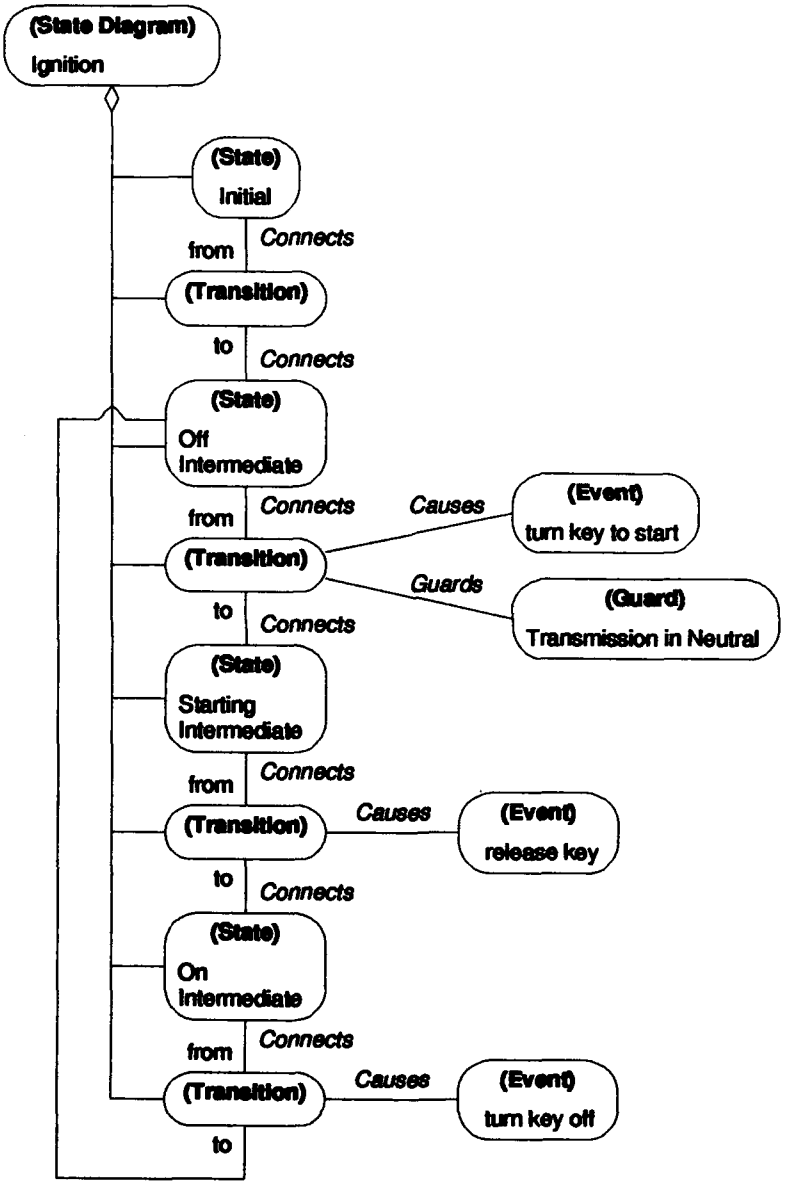


Figure 60. Car Ignition Instance Diagram

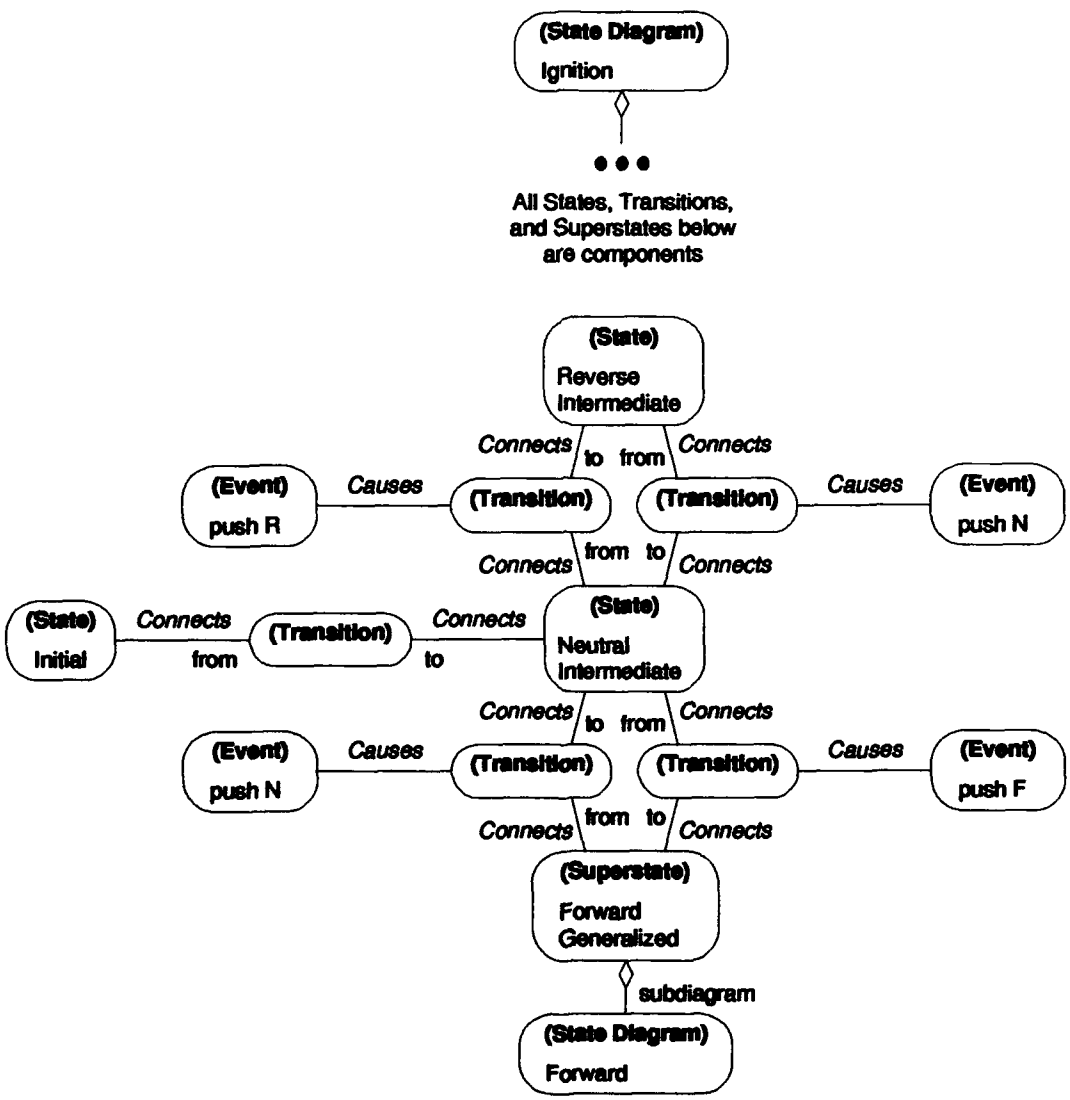


Figure 61. Car Transmission Instance Diagram

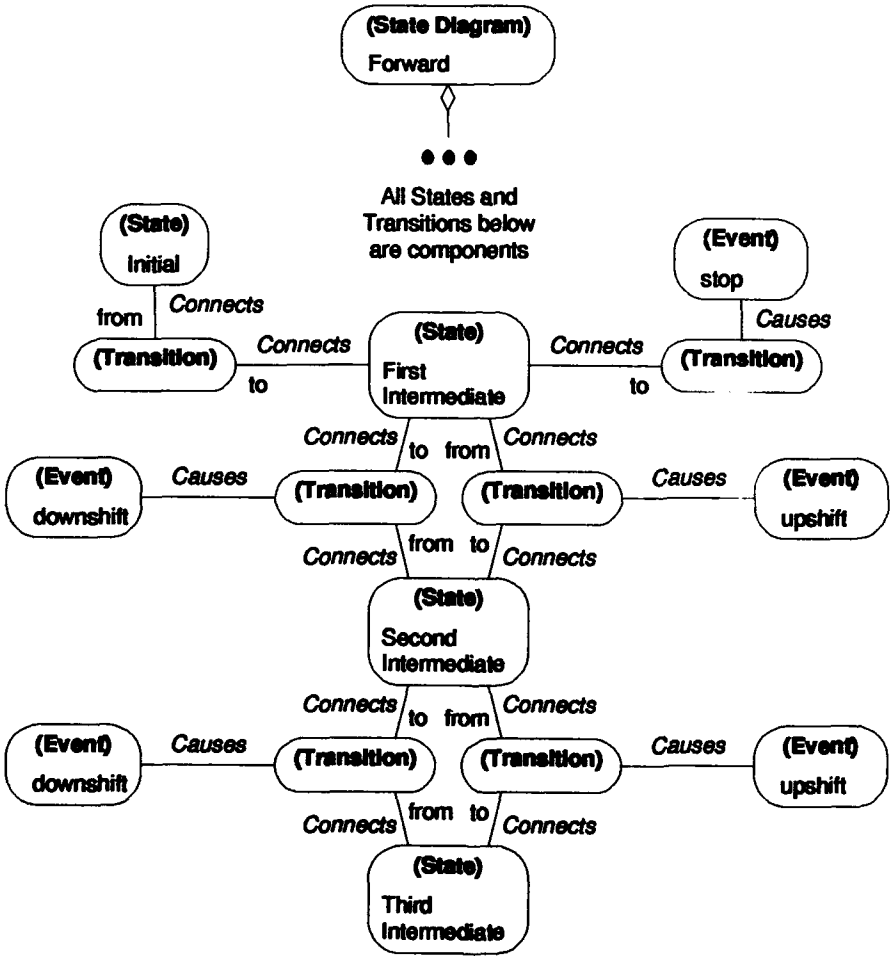


Figure 62. Car Forward Instance Diagram

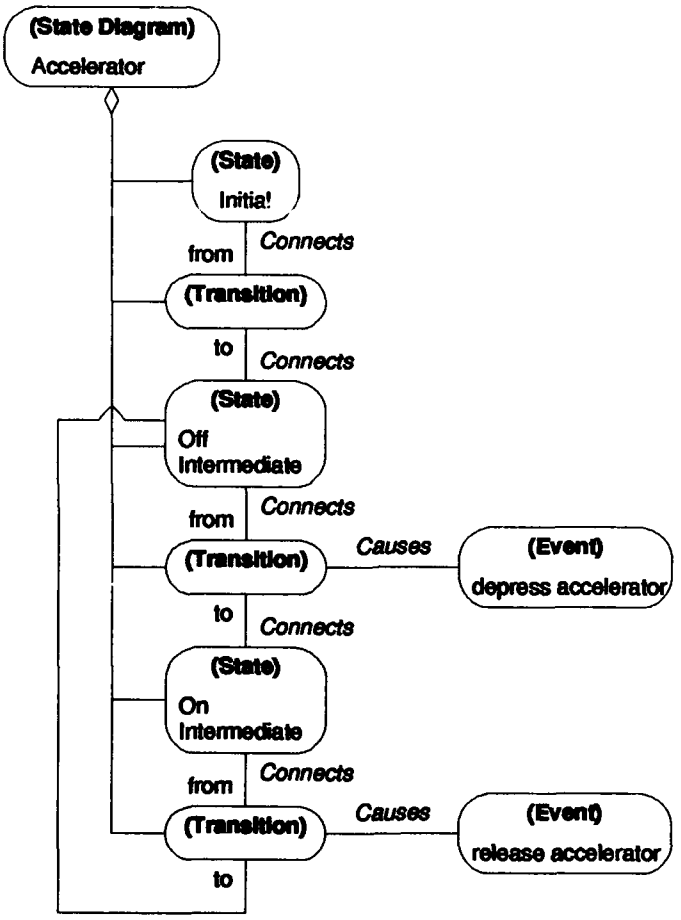


Figure 63. Car Accelerator Instance Diagram

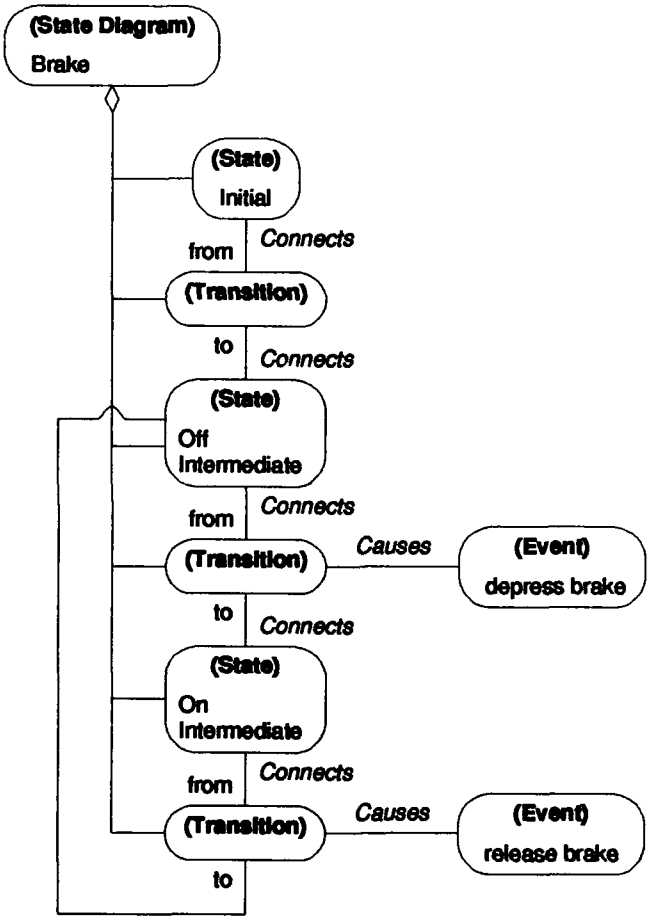


Figure 64. Car Brake Instance Diagram

N.3.3 Banking System Instance Diagram Portions of functional models of a banking system found in Rumbaugh's text are instantiated in this section (17:128,130). Figure 65 shows two data flow models of portions of a banking system. Figures 66–67 show the full instantiation of these data flow model. These diagrams show examples of all the functional model elements except data flow splitting (duplication, composition, decomposition) and process "bubble" expansion (leveling). The following data model design modules are used:

- Data Model Design - see Figure 15
- Flow Connection Design - see Figure 29
- Data Flow Connection Design - see Figure 30
- Data Flow Diagram Design - see Figure 31

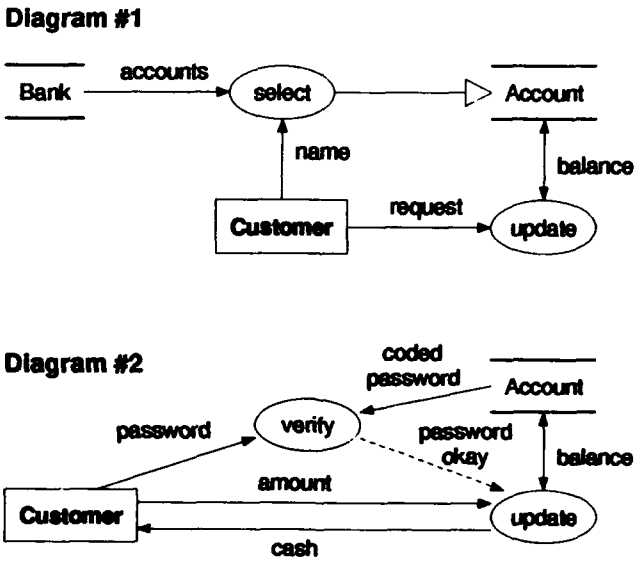


Figure 65. Banking System Data Flow Diagrams

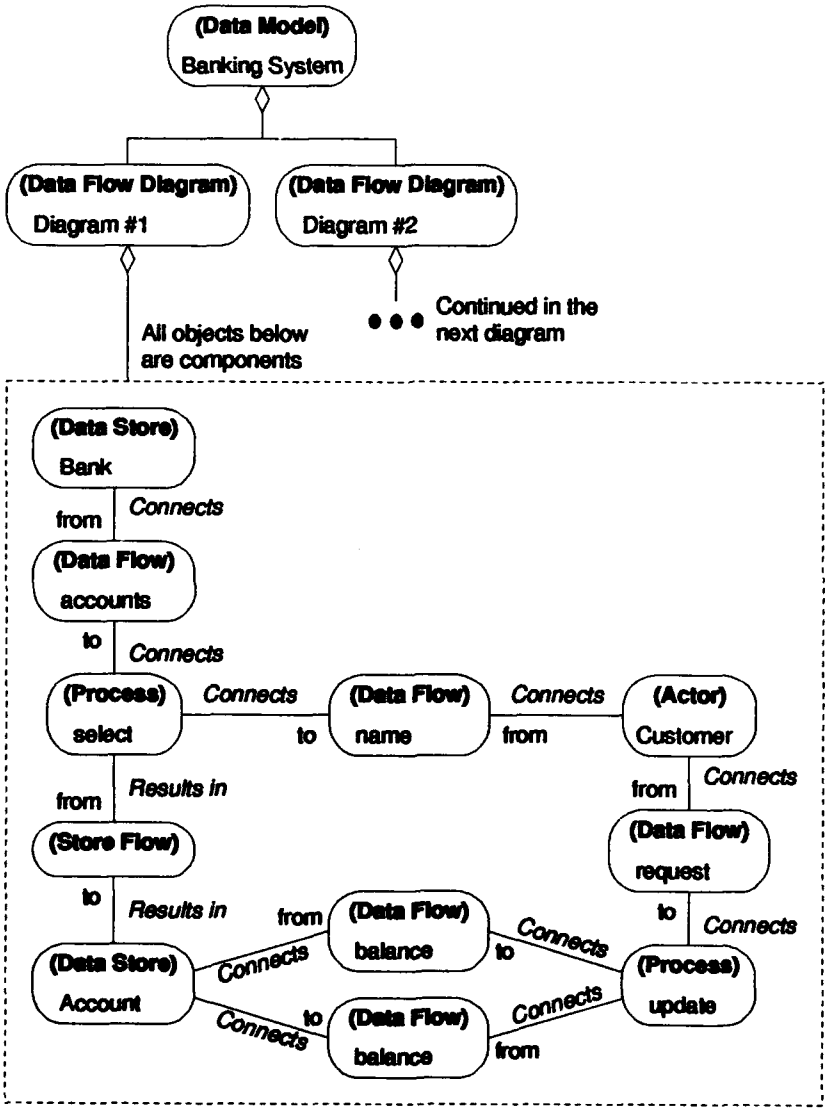


Figure 66. Banking System Instantiation Diagram No.1

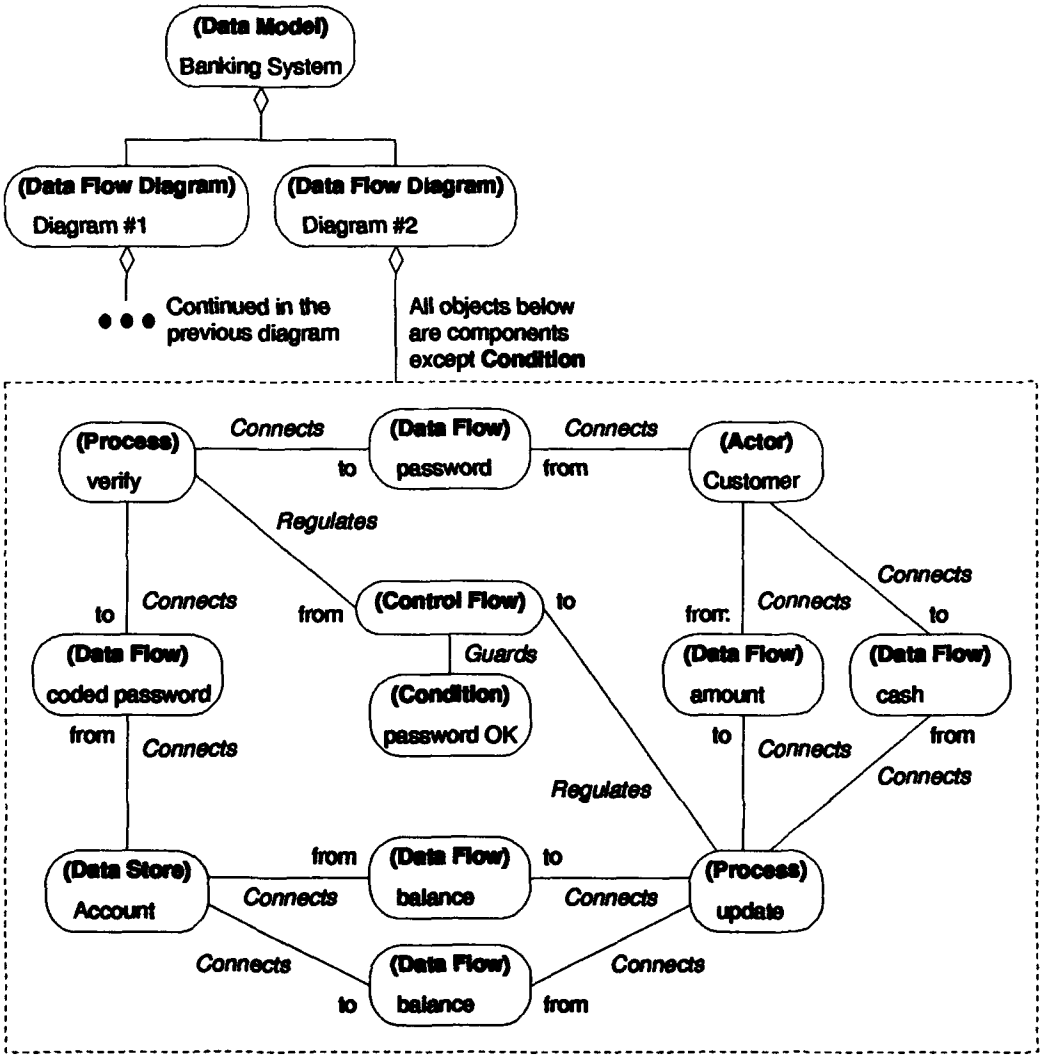


Figure 67. Banking System Instantiation Diagram No.2

Appendix O. *Configuration and User's Guide*

This configuration and user's guide describes the software that was developed for this thesis. It outlines the general organization of the software, the file names and directory structure, how to compile and link the code, and how to use the *driver* program to create and populate the object data repository.

O.1 Software Organization

The software used for this thesis was developed on a Sun SPARCstation. The operating system was SunOS UNIX and the user interface was Open Windows. The source code was written in standard Ada and was compiled with the Verdix Ada Version 6 compiler. No attempts were made to compile the source code on other machines and operating systems. However, an attempt was made to compile the source code using Verdix Ada Version 5. Under Version 5, all source files had to be located in the same directory as the Ada library in order to successfully compile. Version 5 did not provide the -L option to specify an alternate directory for the Ada library. It automatically assumed the Ada library was in the same directory as the source code. After co-locating all source files in the Ada library directory, the code compiled successfully under Version 5.

The directory structure for this thesis project is shown in Figure 68. The main directory contains the Ada library files and subdirectories. Before attempting to compile any source code, the Ada library must be created. The next section describes how to build an Ada library using Verdix Ada Version 6. To build an Ada library with another compiler or version, consult the compiler user's guide. The main directory also contains a shell script file called ***build.sh*** to build the source code. Finally, the ***data_model***, ***drawing_model***, and ***library*** subdirectories contain the source files for this thesis. Because the drawing model was not implemented, there is no code in the ***drawing_model*** subdirectory.

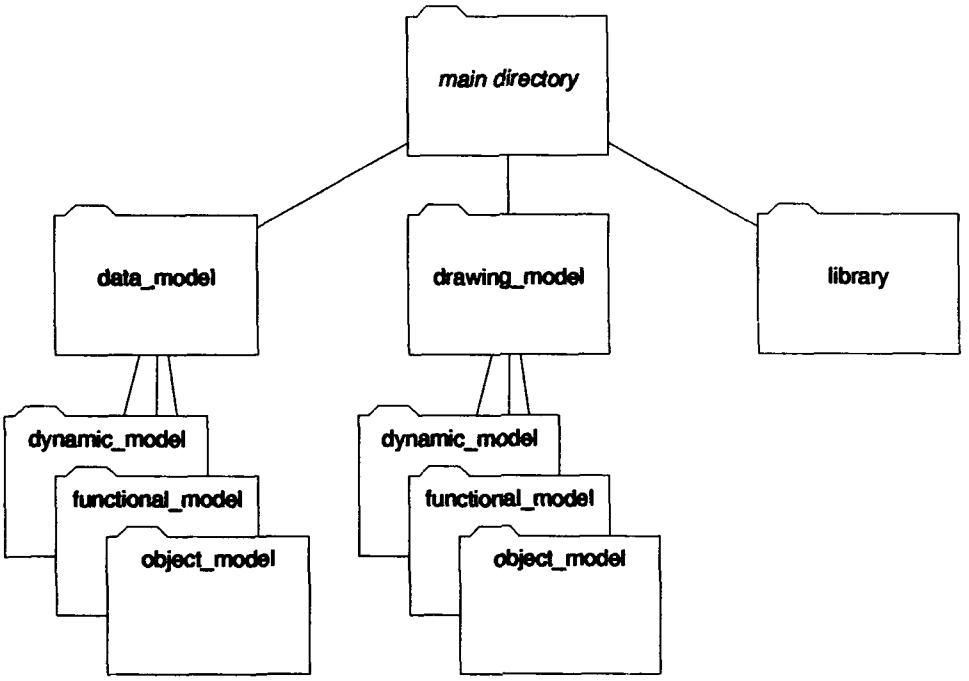


Figure 68. Code Directory Structure

The names of the source files are fairly descriptive. All classes, associations, and aggregations were implemented as classes, and each is contained in its own Ada package. The file names for these packages are the same as the package name with the addition of the required Ada *.a* extension. The general layout of the file name for classes, associations, and aggregations is *CLASS_NAME_class.a*, *ASSOCIATION_NAME_association.a*, and *AGGREGATION_NAME_aggregation.a* respectively.

The ***data_model*** directory contains all the class, association, and aggregation Ada packages supporting the data model. This directory is further divided into three subdirectories in correlation to the three submodels of the OMT. These subdirectories are ***object_model***, ***dynamic_model***, and ***functional_model***. Ada packages for the three submodels are contained within their respective subdirectories. All source code implementing the cross-links between the submodels are contained in the ***data_model*** directory.

The ***data_model*** directory also contains the source code for the ***dispatcher*** and the ***driver***. A shell script file named ***build.sh*** contains the commands needed to compile and link all the modules

in the data model. A later section in this appendix describes how to use the *driver* program once it is created.

The *library* directory contains all the source code needed to support the implementation of the data model. A shell script file named *build.sh* contains the commands needed to compile the *library* source files in the correct sequence. Other files in the *library* directory include packages such as list components, variable-length strings, and object management.

The next section contains a complete list of all the files.

O.2 File and Directory Listing

```
/  (main source directory)
   build.sh
   /data_model
      build.sh
      corresponds_to_association.a
      data_model_aggregation.a
      data_model_class.a
      data_pkg.a
      dispatch_pkg_body.a
      dispatch_pkg_spec.a
      driver.a
      /dynamic_model
         action_class.a
         activity_class.a
         causes_association.a
         condition_class.a
         connects_trans_association.a
         controls_association.a
         conveys_association.a
         event_class.a
         expands_to_std_association.a
         generates_association.a
         guards_association.a
         inherits_event_association.a
         outputs_association.a
         performs_association.a
         sends_to_association.a
         split_class.a
         state_class.a
         state_diagram_aggregation.a
         state_diagram_class.a
         superstate_aggregation.a
         superstate_class.a
         synchronization_class.a
         transition_class.a
         traps_performs_association.a
      /functional_model
         actor_class.a
         composition_class.a
         connects_df_association.a
         control_flow_class.a
         data_flow_class.a
         data_flow_diagram_aggregation.a
         data_flow_diagram_class.a
         data_store_class.a
         decomposition_class.a
         duplication_class.a
         expands_to_dfd_association.a
         process_class.a
         regulates_association.a
         results_in_association.a
         store_flow_class.a
      /object_model
         argument_class.a
         association_aggregation.a
         association_class.a
         attribute_class.a
         behavior_class.a
         class_aggregation.a
         class_class.a
         connection_aggregation.a
         connection_class.a
         connects_conn_association.a
         constrains_association.a
         defines_association.a
         derives_from_association.a
         discriminates_association.a
         domain_class.a
         inheritance_aggregation.a
         inheritance_class.a
         instantiates_association.a
         link_aggregation.a
         link_class.a
         modeled_as_association.a
         module_aggregation.a
         module_class.a
         multiplicity_class.a
         object_aggregation.a
         object_class.a
         object_diagram_aggregation.a
         object_diagram_class.a
         operation_aggregation.a
         operation_class.a
         propagates_association.a
         qualifies_association.a
         role_class.a
         sheet_aggregation.a
         sheet_class.a
         signature_aggregation.a
         signature_class.a
         value_class.a
   /drawing_model
      /dynamic_model
      /functional_model
      /object_model
   /library
      build.sh
      indexed_sequential_list_pkg_body.a
      indexed_sequential_list_pkg_spec.a
      keyed_sequential_list_pkg_body.a
      keyed_sequential_list_pkg_spec.a
      label_pkg.a
      object_pkg_body.a
      object_pkg_spec.a
      sequential_list_pkg_body.a
      sequential_list_pkg_spec.a
      vstrings_body.a
      vstrings_spec.a
```

O.3 Compile and Link Guide

This section describes the steps needed to compile and link the source code listed in the previous section. The general procedure is broken into three main steps: build the Ada library, compile the source code, and link the object code to create the *driver*. The following paragraphs describe these steps in detail.

To build an Ada library in Version 6 of Verdix Ada, do the following:

1. Enter *cd name* to change to the main source code directory.
2. Enter *set path = ($path verdix-directory)* to put the Verdix Ada compiler directory in the PATH environment variable.
3. Enter *a.mkdir* to create the Ada library.

Once the Ada library is built, the code can be generated by running the *build.sh* shell script in the main directory.

1. Enter *cd name* to change to the main source code directory.
2. Enter *set path = ($path verdix-directory)* to put the Verdix Ada compiler directory in the PATH environment variable.
3. Enter *build.sh* to execute the build script.

Depending on the system load, this script will take about one hour to complete.

The following text shows the contents of the *build.sh* scripts.

```
build.sh:
   library/build.sh
   data_model/build.sh

library/build.sh:
   ada library/vstrings_spec.a
   ada library/vstrings_body.a
   ada library/label_pkg.a
   ada library/sequential_list_pkg_spec.a
   ada library/sequential_list_pkg_body.a
   ada library/indexed_sequential_list_pkg_spec.a
   ada library/indexed_sequential_list_pkg_body.a
   ada library/object_pkg_spec.a
   ada library/object_pkg_body.a

data_model/build.sh:
   ada data_model/object_model/*.a
   ada data_model/dynamic_model/*.a
   ada data_model/functional_model/*.a
   ada data_model/*_class.a
   ada data_model/*_aggregation.a
   ada data_model/*_association.a
   ada data_model/data_pkg.a
   ada data_model/dispatch_pkg_spec.a
   ada data_model/dispatch_pkg_body.a
   ada data_model/driver.a
   a.ld -o data_model/driver driver
```

O.4 Driver Program Guide

After compiling and linking the source code as described in the previous section, the *driver* program will be created in the *data_model* directory. This section explains how to use the *driver* program to create and populate the data model.

1. Before running the *driver* for the first time, create a subdirectory to store your copy of the data model. All files needed by the *driver* will be created and stored in this subdirectory. If you want other copies or versions of the data model, create additional directories.
2. Type *cd data_model* to change to the data model directory.
3. Type *driver* to start the driver program.
4. The driver title will be displayed. Afterwards, the program will prompt you to enter a project name. Type the name of one of the subdirectories you created above, followed by a slash (/). The driver will then create and open all the necessary files in this subdirectory. The names and descriptions of these files are as follows:
   - *.OBJECT_NAME* - These files contain the objects created for each class, association, and aggregation in the data model.

- *.master_list* - This file contains pointers to all the objects in the project, that is, all the objects in the *.OBJECT_NAME* files.
- *.next_handle* - This file is used by the object manager to maintain object uniqueness by storing the next available object handle.

5. Two items will be displayed—the current project and the current object. Initially, the current object is blank.
6. The driver then prompts you to enter a command. Immediately above this prompt is a list of available commands. These commands are:

   - PROJECT - change the name of the current project
   - OBJECT - change the name of the current object class
   - CLEAR - clear or delete all the objects
   - CREATE - create a new object
   - DELETE - delete an object
   - GET - get or retrieve the attributes of an object
   - SET - set the attributes of an object
   - LIST - list the object handles
   - LOAD - load the objects into memory
   - SAVE - save the objects to disk
   - QUIT - quit the driver program

7. Enter *load* to load the data model into memory.
8. You can now run any of the other commands (described below). Before quitting, enter *save* to save your changes; otherwise, your changes will be lost.
9. Enter *quit* to exit the driver program.

The *driver* commands operate in one of two modes: with or without a current object selected. When an object is selected, commands operate only on the current object list. When an object is not currently selected, commands operate on the entire (master) list of objects.

The *driver* program is not very smart—it does exactly what you tell it! It will not inform you if you overwrite files or forget to save before exiting. Furthermore, it cannot recover if you "ungracefully" exit the program such as pressing CONTROL-C. Therefore, care should be taken to prevent logical and unexpected errors in the data.

Error messages occur when you attempt to do any of the following:

- Enter an invalid command
- Select a non-existing object class name
- Enter a non-existing object handle

Other errors are not handled and will cause the program to crash with an exception error.

The following subsections individually describe each of the *driver* commands.

O.4.1 PROJECT Command The purpose of the PROJECT command is to change the current project. This command is called automatically when the *driver* is initially started. This command will not automatically load the objects into memory.

1. Enter *project* at the command prompt to activate this command.
2. The project name prompt will display. Enter the name of the project directory followed by a slash (/). You can also add an optional project name suffix to create more than one project in the same project directory. For example, you could enter *test/1* to use the data model from *test-1* project, or you could enter *test/2* to use the data model from *test-2* project.
3. If this is a new project, the necessary project files will then be created; otherwise, the existing project files will be opened.
4. Run the LOAD command, if needed, to load the objects into memory.

O.4.2 OBJECT Command The purpose of the OBJECT command is to change the current object class name. When the *driver* is initially started, the current object is blank.

1. Enter *object* at the command prompt to activate this command.
2. The object name prompt will display. Enter the name of any object class. Valid object class names can be found by listing the files in the project directory. For example, the *activity* class name is *activity_class*, the *class* aggregation name is *class_aggregation*, and the *constrains* association name is *constrains_association*. Blanks can be used in place of underscores (_) when entering the object name.
3. All future commands will now operate on the new current class just entered.
4. If you want to blank out the current class in order to allow commands to operate on any or all object classes, just press return when you are prompted to enter an object name.

O.4.3 CLEAR Command The purpose of the CLEAR command is to clear or delete objects in memory. If a current object class is selected, this command will clear all objects in that class; otherwise, all objects in all classes with be deleted.

1. Run the OBJECT command if you want to clear only one class of objects.
2. Enter *clear* at the command prompt to activate the CLEAR command.
3. A message will display informing you that the objects are being cleared. When completed, the command prompt will be redisplayed.

O.4.4 CREATE Command The purpose of the CREATE command is to create an object in memory.

1. Enter *create* at the command prompt to activate this command.
2. If a current object is not selected, the OBJECT command will run first, and you will be prompted to provide an object name.
3. After an object is created in the current class, the new object handle will be displayed, and the command prompt will be redisplayed.

O.4.5 DELETE Command The purpose of the DELETE command is to delete an object from memory.

1. Enter *delete* at the command prompt to activate this command.
2. The object handle prompt will then be displayed. Enter the handle of an existing object. If a current object class is selected, then you must choose a handle from that class; otherwise, you can enter any existing handle.
3. The object will be deleted if it exists, and the command prompt will be redisplayed.

O.4.6 GET Command The purpose of the GET command is to get or query the attribute values of an existing object in memory.

1. Enter *get* at the command prompt to activate this command.
2. The object handle prompt will then be displayed. Enter the handle of an existing object. If a current object class is selected, then you must choose a handle from that class; otherwise, you can enter any existing handle.
3. The attribute names and values will then be displayed, followed by the command prompt.

O.4.7 SET Command The purpose of the SET command is to set the attribute values of an existing object in memory.

1. Enter *set* at the command prompt to activate this command.
2. The object handle prompt will then be displayed. Enter the handle of an existing object. If a current object class is selected, then you must choose a handle from that class; otherwise, you can enter any existing handle.
3. Each of the attribute names will be displayed one-by-one as you are prompted to enter values for these attributes. When completed, the command prompt will be redisplayed.

O.4.8 LIST Command The purpose of the LIST command is to list all object handles. If a current object class is selected, this command will list all object handles in that class; otherwise, all object handles in all classes with be listed.

1. Run the OBJECT command if you want to list only one class of objects.
2. Enter *list* at the command prompt to activate this command.
3. If a current object name is selected, only the handles in that class are listed or displayed; otherwise, the handles are displayed along with the name of the class they are in.

O.4.9 LOAD Command The purpose of the LOAD command is to load all objects into memory. The LOAD command will overwrite any previous objects in memory. If a current object class is selected, this command will load all objects in that class; otherwise, all objects in all classes with be loaded.

1. Run the OBJECT command if you want to load only one class of objects.
2. Enter *load* at the command prompt to activate this command.
3. After loading the objects, the command prompt will be redisplayed.

O.4.10 SAVE Command The purpose of the SAVE command is to save all objects to disk. The SAVE command will overwrite any previous objects saved on disk. If a current object class is selected, this command will save all objects in that class; otherwise, all objects in all classes with be saved.

1. Run the OBJECT command if you want to save only one class of objects.
2. Enter *save* at the command prompt to activate this command.
3. After saving the objects, the command prompt will be redisplayed.

O.4.11 QUIT Command The purpose of the QUIT command is to exit the *driver* program. This command will not automatically save any changes you made to the object lists in memory.

1. Run the SAVE command if you want to save changes to disk.
2. Enter *quit* at the command prompt to activate the QUIT command.
3. The program will exit, and the shell prompt will reappear.

Bibliography

1. Ada 9X Mapping/Revision Team, Intermetrics, Inc., Cambridge, Massachusetts. *Ada 9X Mapping: Mapping Rationale*, version 4.1 edition, March 1992.
2. Bowles, Adrian. Object-slicing–it's as dangerous as it sounds. *Object Magazine*, 2(1):25–26, May/June 1992.
3. Meyer, Bertrand. Genericity versus inheritance. *Journal of Pascal, Ada, and Modula-2*, 7(2):20–21, March/April 1988.
4. Stinson, Craig. Microsoft says ole. *PC Magazine*, 10(5):34, March 1991.
5. Mandelkern, Dave. Visual programming: Interactive construction of programs speeds development and increases productivity. *Object Magazine*, 2(3):39–43, September/October 1992.
6. Harel, David. Biting the silver bullet: Toward a brighter future for system development. *IEEE Computer*, pages 8–20, January 1991.
7. Harel, David and Doron Drusinsky. Using statecharts for hardware description and synthesis. *IEEE Transactions on CAD*, 8(2):798–807, July 1989.
8. de Champeaux, Dennis and Penlope Faure. A comparative study of o-o analysis methods. *Journal of Object-Oriented Programming*, 5(1):21–33, March/April 1992.
9. Jones, Do-While. Just what are the rules of inheritance anyway? *Journal of Pascal, Ada, and Modula-2*, 8(3):72–75, May/June 1989.
10. Ullman, Ellen. Oop made visual: Digitalk's look and feel kit. *Byte*, 16(8):112–113, August 1991.
11. General Electric Concepts Center. *OMTool Product Literature*, 1992.
12. Booch, Grady. *Software Engineering with Ada*. Benjamin-Cummings, second edition, 1986.
13. Booch, Grady. *Object-Oriented Design with Applications*. Benjamin/Cummings, Redwood City, 1991.
14. Edwards, Lacy H. The promise of ot: You should be smiling by now. *Object Magazine*, 2(3):15, September/October 1992.
15. Fersko-Weiss, Henry. Symbologic's adept builds expert systems graphically. *PC Magazine*, 10(22):58, December 1991.
16. Jacobson, Ivan. Panel: Structured analysis and object-oriented analysis. In *ACM SIGPLAN Notices: OOPSLA 1990 Proceedings*, volume 25 number 10, pages 135–, November 1990.
17. Rumbaugh, James et al. *Object-Oriented Modeling and Design*. Prentice-Hall, Englewood Cliffs, NJ, 1991.
18. Mark IV Systems, Inc. *Object Maker User's Manual*, 1991.
19. Blaha, Michael. Models of models. *Journal of Object-Oriented Programming*, 5(5):13–18, September 1992.
20. Coad, Peter and Edward Yourdon. *Object-Oriented Analysis*. Prentice-Hall, Englewood Cliffs, NJ, second edition, 1991.
21. Coffee, Peter. Digitalk's parts is the definitive 'visual' development tool for os/2. *PC Week*, 9(41):95–97, October 12 1992.
22. Kendall, Robert. Icon author: Multimedia authoring made (just a bit) easier with innovative flowchart metaphor. *PC Magazine*, 10(21):48, December 1991.

23. Miller, Rock. Object vision builds smarter forms with visual programming. *PC Magazine*, 10(19):413, November 1991.

24. Ahmed, Shamim et al. Object-oriented database management systems for engineering: A comparison. *Journal of Object-Oriented Programming*, 5(3):27+, June 1992.

25. Software Productivity Consortium, Herndon, Virginia. *Ada Quality and Style: Guidelines for Professional Programmers*, 02.00.02 edition, 1991.

26. Software Productivity Solutions, Inc., Indialantic, Florida. *Classic Ada User's Manual*, 9.0 edition, 1992.

27. Bach, William W. Is ada really an object-oriented programming language? *Journal of Pascal, Ada, and Modula-2*, 8(2):18–25, March/April 1989.

Vita

Captain Stephen P. Perucca was born on 23 September 1957 in Denver, Colorado. He graduated from Charles M. Russell High School in Great Falls, Montana in 1975 with honors in Math, and afterwards, he worked as an RPG-II computer programmer for Western Computer Services in Great Falls. He attended Brigham Young University in Provo, Utah and graduated in June 1987 with a Bachelor of Science degree in Computer Science. Captain Perucca attended the 49xx Officer Technical Training Course at Keesler AFB, Mississippi and graduated in March 1988. He was then assigned to the 7th Communications Group (7CG), Pentagon as a 4925 officer and was tasked to provide computer systems support for the Office of the Assistant Secretary of Defense (OASD), Reserve Affairs (RA). For two and a half years, Captain Perucca's duties at RA involved systems management, network management, hardware installation, systems programming, software evaluation, and user support and training. He was then assigned to the 7CG Security Directorate, Computer Security Branch as the Chief of Computer Security where he oversaw computer security for over one billion dollars worth of mainframe computers and provided expert computer security guidance to the 7CG commander. In April 1991, Captain Perucca was selected to attended the Air Force Institute of Technology to persue a Masters of Science degree in Computer Science.

Permanent address: 738 33B Avenue NE
Great Falls, Montana 59404

REPORT DOCUMENTATION PAGE

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1992	Master's Thesis

4. TITLE AND SUBTITLE
ADA IMPLEMENTATION OF AN OBJECT DATA REPOSITORY

6. AUTHOR(S)
Stephen P. Perucca, Capt, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GCS/ENG/92D-11

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Captain Phil Lienert
ASC/RWWW
Wright-Patterson AFB, Ohio

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

The many benefits of object-oriented software development such as encapsulation and extendibility have inspired numerous models of the object-oriented paradigm. Rumbaugh's Object Modeling Technique (OMT) is an object-oriented model that uses three submodels. The object, dynamic, and functional submodels of the OMT describe the data, behavioral, and processing aspects of a system by using entity-relationship, state-transition, and data flow models. Cross-links relate how the three submodels tie together. Two metamodels (models of models) of the OMT are developed using the OMT methodology and notation. The essential data elements of the OMT are abstracted into a data metamodel, and the graphical elements are abstracted into a drawing metamodel. Visual programming concepts and examples are briefly discussed. The OMT model is analyzed and designed using OMT object models. The data and drawing elements are modeled and implemented in standard Ada as object classes, associations, and aggregations. An object manager is developed to provide a generic core class, to maintain an object data repository, and to assert unique object identities. Instantiated examples (instance diagrams) verify the correctness of the metamodel designs. Problems encountered during development are discussed and recommendations are made to improve the OMT. Possible future research areas are presented.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Ada, Object-Oriented, Object Management, Metamodeling, Object Modeling, Object Models, Object Database, Object Repository, Visual Programming	350
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18
298-102